НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:

Завідувач кафедри

_____ Сергій СТІРЕНКО

«___»_____2020 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Комп'ютерні системи та мережі»

спеціальності 123 «Комп'ютерна інженерія»

на тему: «Спосіб конструювання трафіку з врахуванням рівномірного завантаження мережі»

Виконав:
студент IV курсу, групи ІО-62
Савченко Юрій Юрійович _____

Керівник:
Асистент Калюжний Олександр Олегович _____

Консультант з нормо контролю:
Проф. д.т.н. Сімоненко Валерій Павлович _____

Рецензент:
Доц. каф. СПіСКС к.т.н., доц.
Орлова Марія Миколаївна _____

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.

Студент _____

Київ – 2020 року

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМ. ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 123 «Комп'ютерна інженерія»

Освітньо-професійна програма «Комп'ютерні системи та мережі»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ *Сергій СТІРЕНКО*
(підпис)
"___"_____20__ р.

ЗАВДАННЯ

на дипломний проєкт студенту

Савченку Юрію Юрійовичу

1. Тема проєкту «*Спосіб конструювання трафіку з врахуванням рівномірного завантаження мережі»*
керівник проєкту Калюжний Олександр Олегович, асистент, затверджені наказом по університету від *«*07*»* травня __2020р. №__1081-с__
2. Термін подання студентом проєкту_____2020р.
3. Вихідні дані до проєкту технічне завдання, теоретичні дані.
4. Зміст пояснювальної записки: *Дослідження і вивчення запропонованих рішень. Вибір технологій розробки. Реалізація запропонованого алгоритму. Розробка архітектури програми. Розробка відповідної програми моделювання алгоритму.*

5. Консультант роботи, з вказівкою розділів роботи, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормо контроль	Сімоненко В.П.		

6. Дата видачі завдання *01.09 .2019 року*

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Найменування етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту(роботи)	Примітки
1.	Затвердження теми роботи	05.08.2019	
2.	Вивчення та аналіз завдання	10.09.2019-17.02.2020	
3.	Розробка архітектури та загальної структури системи	17.02.2020-29.03.2020	
4.	Розробка структур окремих підсистем	29.03.2020-06.04.2020	
5.	Програмна реалізація системи	06.04.2020-14.04.2020	
6.	Оформлення пояснювальної	14.04.2020-19.05.2020	
7.	Захист програмного продукту	28.05.2020	
8.	Перед захист	02.06.2020	
9.	Захист	25.06.2020	

Студент Юрій САВЧЕНКО _____
(підпис)

Керівник Олександр КАЛЮЖНИЙ _____
(підпис)

Анотація

У даній бакалаврській дипломній роботі було створено власний алгоритм маршрутизації для транспортних мереж. Також було розроблено відповідний інтерфейс користувача для моделювання створеного алгоритму.

Розроблений алгоритм дозволяє виконувати пошук найоптимальніших шляхів в транспортній мережі, тим самим надаючи можливість рівномірного завантаження мережі. Відповідний інтерфейс користувача дозволяє виконувати моделювання розглянутого алгоритму.

Створений програмний продукт надає можливість швидко та ефективно виконувати моделювання алгоритму маршрутизації, що в свою чергу дозволяє в конкретний момент часу виконувати пошук найоптимальнішого маршруту в транспортній мережі. Користувач має доступ до повного контролю мережі, а також може виконувати ряд операцій для моделювання алгоритму та відображення результатів його роботи.

Описаний продукт був створений в текстовому редакторі Sublime Text 3. Для реалізації програмного продукту було використано мову програмування C++ та фреймворк Qt.

Annotation

In this bachelor's thesis, the own routing algorithm for transport networks was created. Also, the appropriate user interface for modeling created algorithm was implemented.

The algorithm was created allows finding the most optimal paths in the transport network, thereby providing an ability to make uniform network load. The appropriate user interface allows making modeling of the considered algorithm.

The program product was created gives an ability fast and effective to make the modeling of the routing algorithm, which allows making a finding of the most optimal path in the transport network at a particular time moment. The user has full access to control the network and also can make a row of operations for modeling of algorithm and demonstrating results of its work.

The reviewed product was created in the Sublime Text 3 text editor. The programming language C++ and framework Qt were used to implement the program product.

ВІДОМІСТЬ ДИПЛОМНОГО ПРОЄКТУ

№ з/п	Формат	Позначення	Найменування	Кількість листів	Примітка
1	А4		Завдання на дипломний проєкт	2	
2	А4	ІАЛЦ.466454.001 ВП	Відомість проєкту	1	
3	А4	ІАЛЦ.466454.002 ТЗ	Технічне завдання	4	
4	А4	ІАЛЦ.466454.003 ПЗ	Пояснювальна записка	62	
5	А3	ІАЛЦ.466454.004 Д1	Принципова схема алгоритму	1	
6	А3	ІАЛЦ.466454.005 Д2	Функціональна схема(діаграма класів)	1	
7	А3	ІАЛЦ.466454.006 Д3	Структурна схема реалізованого застосунку	1	
8	А4	ІАЛЦ.466454.007 Д4	Текст програми	23	

Зм.	Арк.	№ документна	Підпис	Дата	ІАЛЦ.466454.001 ВП		
Розроб.		Савченко Ю.Ю.			Спосіб конструювання трафіку з рівномірним завантаженням мережі	Літ.	Аркуш 1 / Акрушів 1
Перевір.		Калюжний О.О.					
Н. Контр.		Сімоненко В.П.				НТУУ «КПІ ім. Ігоря Сікорського», ФІОТ, гр.ІО-62	
Затверд.		Стіренко С.Г.					

ТЕХНІЧНЕ ЗАВДАННЯ

до дипломної роботи

освітньо-кваліфікаційного рівня__бакалавр___

на тему: “ Спосіб конструювання трафіку з врахуванням рівномірного завантаження мережі”

Київ – 2020 року

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ 2
2. ПІДСТАВИ ДЛЯ РОЗРОБКИ 2
3. МЕТА І ПРИЗНАЧЕННЯ РОЗРОБКИ 2
4. ДЖЕРЕЛА РОЗРОБКИ 2
5. ТЕХНІЧНІ ВИМОГИ 3
   - 5.1. Вимоги до продукту, що розробляється 3
   - 5.2. Вимоги до програмного забезпечення 3
   - 5.3. Вимоги до апаратної частини 3

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку і в майбутньому подальшу підтримку запропонованого програмного продукту ”Програмний застосунок для моделювання алгоритму маршрутизації”.

Область застосування даного програмного забезпечення – використання як засобу виконання пошуку найоптимальніших маршрутів в транспортних мережах з забезпеченням рівномірного завантаження мережі.

2. ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки даного продукту є завдання для виконання роботи кваліфікаційно-освітнього рівня «бакалавр комп’ютерної інженерії», який був затверджений факультетом “Інформатики та обчислювальної техніки” кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

3. МЕТА І ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою і призначенням написання даного програмного продукту є ідея створення власного програмного продукту для моделювання алгоритму маршрутизації в транспортних мережах.

4. ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки дипломного проєкту є науково-технічна література з теорії графів та мереж, наукові статті, публікації в Інтернеті з питань маршрутизації.

5. ТЕХНІЧНІ ВИМОГИ

5.1.Вимоги до продукту, що розробляється

- Надати програмному застосунку необхідний функціонал для моделювання алгоритму маршрутизації;
- Надати можливість зчутувати та записувати відповідний файл з конфігурацією топології мережі;
- Надати програмного застосунку можливість взаємодії з користувачем за допомогою зручного графічного інтерфейсу та команд користувача;
- Надати програмному застосунку можливість відображення знайденого найоптимальнішого маршруту та завантаження мережі після пошуку;
- Забезпечити програмний застосунок можливістю відображення відповідного вікна аналізу знайдених результатів за допомогою гістограм.

5.2.Вимоги до програмного забезпечення

- Будь-яка операційна система з підтримкою графічного інтерфейсу користувача;
- C++-11, Qt5.13.0;
- Sublime Text3, Qt Creator;
- Бібліотека QCustomPlot.

5.3.Вимоги до апаратної частини

- ЦП не менше ніж Intel(R) Celeron(R) 420;
- ROM не менше ніж 100 ГБ;
- RAM не менше ніж 4 ГБ.

ЕТАПИ РОЗРОБКИ

	Дата
Аналіз і вивчення літератури	17.11.2019
Складання технічного завдання	15.12.2019
Дослідження та вибір ресурсів для розробки	27.01.2020
Проєктування архітектури програмного забезпечення	24.02.2020
Розробка програмного забезпечення	06.04.2020
Тестування програмного забезпечення	09.04.2020
Налагодження та виправлення помилок	10.04.2020
Написання документації дипломного проєкту	03.05.2020

Пояснювальна записка

до дипломного проєкту

на тему: «*Спосіб конструювання трафіку з рівномірним завантаженням мережі*»

Київ – 2020 року

ЗМІСТ

ВСТУП 2
РОЗДІЛ 1 ОГЛЯД ТРАНПСПОРТНИХ МЕРЕЖ 3
1.1. Транспортна мережа 3
1.2. Модель транспортної мережі 4
1.3. Маршрутизація в транспортних мережах 8
1.4. Порівняння алгоритмів аналогів 12
ВИСНОВКИ ДО РОЗДІЛУ 1 17
РОЗДІЛ 2 АЛГОРИТМ МАРШРУТИЗАЦІЇ 18
2.1. Опис основних етапів алгоритму 18
2.2. Формування множини шляхів 19
2.3. Приклад роботи алгоритму 20
2.4. Порівняння пошуку шляхів від зміни динаміки мережі 33
ВИСНОВКИ ДО РОЗДІЛУ 2 38
РОЗДІЛ 3 МОДЕЛЮВАННЯ ЗАПРОПОНОВАНОГО АЛГОРИТМУ 39
3.1. Огляд технологій 39
3.2. Опис логіки програми 40
ВИСНОВКИ ДО РОЗДІЛУ 3 46
РОЗДІЛ 4 ОГЛЯД РОЗРОБЛЕНОГО ЗАСТОСУНКУ 47
4.1. Опис програмного інтерфейсу 47
4.2. Розгляд прикладу роботи програми 56
ВИСНОВКИ ДО РОЗДІЛУ 4 60
ВИСНОВКИ 61
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ 62

ВСТУП

У сьогоднішній день технології не стоять на місці і все з плином часу змінюється та модернізується. Усе контролюється надзвичайно великим потоком даних, що в свою чергу підконтрольні новітнім технологіям. Надзвичайно важливою є задача маршрутизації, не тільки в інтернет мережах, а й в тих, що відносяться до транспортних, адже потреба в виборі найоптимальнішого шляху між містами є однією із основних для власників транспорту та різного бізнесу, що базується на використанні транспорту. Ще однією із основних потреб є вирішення задачі маршрутизації в заторах, оскільки транспорт ключовий спосіб переміщення містом або ж країною. Тому визначення найоптимальнішого маршруту є надзвичайно важливою частиною міського та міжміського пересування.

Основним підходом до вирішення задачі маршрутизації для транспортних мереж є створення власного та швидкісного алгоритму, що в свою чергу дозволить за константний час знаходити найкоротший та найоптимальніший маршрут в заданій транспортній мережі, в певний момент часу. Такий алгоритм можна віднести до алгоритмів реального часу, що повинні вирішувати поставлену задачу за конкретно встановлений час, без затримок та помилок.

Згідно із розглянутими необхідностями виникає потреба у створенні власного алгоритму реального часу, що дозволить вирішувати задачу маршрутизації для транспортних мереж в реальному часі та за константний час. Даний алгоритм повинен бути простим і водночас ефективним у вирішенні поставленої проблеми.

РОЗДІЛ 1
ОГЛЯД ТРАНПСПОРТНИХ МЕРЕЖ

1.1. Транспортна мережа

Транспортна мережа – реалізація спеціалізованої просторової мережі, що використовується для відображення руху транспорту або ж рідного роду вантажів [1]. Прикладом таких мереж можуть бути мережі авто-шляхів, залізничних шляхів, мережі вулиць, трубопроводів та електричних мереж.

Транспортні мережі слугують хорошою можливістю для організації руху транспорту в мережі вулиць, міст та країн. Приклад транспортної мережі продемонстрований на рис. 1.1.

Рис. 1.1. Приклад транспортної мережі

Транспортні мережі у сучасному світі дозволяють регулювати різного роду транспортні перевезення, за допомогою них ми щодня користуємось навігацією навіть про це не задумуючись. Будь-яка навігація а також веб сервіси для доставки товарів та продуктів мають в своєму арсеналі застосунки, що працюють з транспортними мережами для визначення різних маршрутів. Транспортні мережі є основною будь-яких транспортних сервісів.

Переважна більшість існуючих завдань поточного транспортного планування різних перевезень вирішується з застосуванням транспортної мережі або як елемента вказаної системи перевезень, або ж як окремого елемента зовнішнього середовища. Значимість транспортної мережі додатковим чином підкреслена в списках елементів транспортної системи при вирішенні різноманітних задач в зазначеній області організації перевезень і дорожнього руху.

1.2. Модель транспортної мережі

На сьогоднішній день є чимало розроблених моделей моделювання транспортних мереж [2]. Очевидним фактом є те, що під час моделювання моделей транспортних мереж слід враховувати не тільки опис шляхів пересування а й додаткові характеристики для повної моделі. До основних моделей моделювання можна віднести наступні:

1) Координатне моделювання;
2) Топологічне моделювання.

Слід детально розглянути кожен із методів. Для початку необхідно розглянути координатний метод. Координатне моделювання представляє собою розробку цифрового аналога карти як конкретної моделі транспортної мережі, що в свою чергу є доволі зручним для подальшої роботи з мережею. Розроблений повний координатний метод є надзвичайно зручним у використанні. Розгляд об’єкту відбувається на координатній сітці в прямокутній системі координат. Відразу після представлення об’єкта представлена інформація перетворюється з графічної в цифрову та виконується подальша обробка. Перетворення інформації з графічної в цифрову відбувається за допомогою прямокутних матриць, розмір яких повністю відповідає розмірності координатних осей, що представлені в прямокутній системі координат. Дані матриці переважно являють собою булеві матриці, де 1 означає наявність маршруту на відповідній ділянці заданої місцевості, 0 же в свою чергу вказує на те, що будь-який маршрут відсутній

для даної ділянки. Існують також матриці і не булевого формату, котрі представляють позитивними чисельними значеннями обсяги відправки, негативними чисельними значеннями – обсяги споживання вантажу в конкретній точці.

Розглянутий координатний метод є універсальним. Даний метод дає змогу описати надзвичайно велику кількість характеристик транспортної мережі, однак для цього необхідно вводити спеціальні матриці. З іншого ж боку даний метод містить і недоліки. Одним з таких недоліків є методи розрахунків, що в свою чергу є надзвичайно простими в програмній реалізації. Недолік полягає в багаторазовому та циклічному перегляді матриці транспортної мережі для знаходження найкоротших шляхів. Слід наголосити, що кількість таких розрахунків не підлягають скороченню.

Причинами розглянутого недоліку є проблема в моделюванні не тільки тих об’єктів, які відносяться до поставленої задачі, а й низки інших об’єктів, що не використовуються.

Якщо розглядати моделювання транспортних мереж, то існують більш ефективні способи класифікації інформації, що не відноситься до самої мережі. Такий підхід є доречним для обширних характеристик транспортних об’єктів. Такі дані можуть бути описані за допомогою простих лінійних масивів. Зручність використання масивів полягає в тому, що до числа їх внутрішніх характеристик додаються значення координатів обраного об’єкта. Незважаючи на всі переваги розглянутого методу він містить найголовніший недолік, що полягає у відсутності можливості моделювання моделей для магістралей.

Наявність надзвичайно великої кількості інформації перешкоджає підвищенню точності запропонованої моделі. Однак існують різні способи покращення точності. Одним із таких способів є зменшення розмірів клітинки координатної сітки. Зменшення розмірів клітинки призводить до збільшення кількості клітинок та збільшення розмірів і самих матриць. Для прикладу,

двокртане зменшення розміру клітинки викликає чотири-кратне збільшення їх кількості. Такі ж вимоги виникають до пам’яті портативного комп’ютера. Після отриманих скорочень швидкість розрахунків значно сповільнюється, це відбуваються в результаті використання такого способу реалізації.

Іншим і більш зручним та швидшим методом є метод топологічного моделювання. Топологічний метод є більш зручним методом для дослідження об’єктів, що представлені у вигляді багаторівневих систем. Запропонований об’єкт, що підлягає дослідженню представляється у вигляді набору вершин та ланок (ребер). Змодельована транспортна мережа представляється у вигляді графа, що в свою чергу складається з V вершин та E ребер: $G = [V, E]$.

За допомогою вершин можна представляти різні об’єкти транспортної комунікації, іншими словами – пункти переміщення. Для прикладу це можуть бути перехрестя вулиць, основні транспортні пункти, населені пункти, вокзали, тощо. Вершина позначається точкою на графі (розробленій топологічній схемі) змодельованої транспортної мережі.

Ребрами або ж ланками топологічної мережі (графа) описуються маршрути між перерахованими вище транспортними пунктами. Кожне ребро з’єднує два транспортні пункти. Ребро на графі (топологічній схемі) представляється у вигляді відрізка між двома точками.

Сполучення вершин та ланок вважається модельованим об’єктом. Розмір графа визначається кількістю вершин графу. Кожна ланка містить характеристику, яке задається надписом над ланкою. Даною характеристикою можна описати завантаженість представленого маршруту або ж значимість даного шляху.

Топологічна схема транспортної мережі не є кінцевим варіантом моделювання, кінцевим варіантом транспортної мережі є математична модель представлення топологічної схеми. Для отримання математичної моделі існує декілька варіантів представлення, однак однією із найкращих та найзручнішою є матрична модель представлення, що зберігає увесь граф

топологічної схеми у вигляді матриці переходів, або ж матриці пропускних здатностей, що дозволяє швидко та зручно виконувати дії над створеної мережею.

Топологічний метод є одним із найуживаніших для моделювання та аналізі транспортних мереж. Даний метод дозволяє використати ряд переваг. Першим із таких переваг є просте уявлення моделі. Другою перевагою є простота побудови складної моделі, що може складатись із сукупності простіших моделей та об'єднуватись у одну єдину транспортну модель. Третьою перевагою є зручність обчислень над запропонованою моделлю, за рахунок чіткого математичного апарату та відповідних ефективних алгоритмів обчислювальної техніки.

Приклад графу або ж топологічної схеми продемонстрований на рис. 1.2.

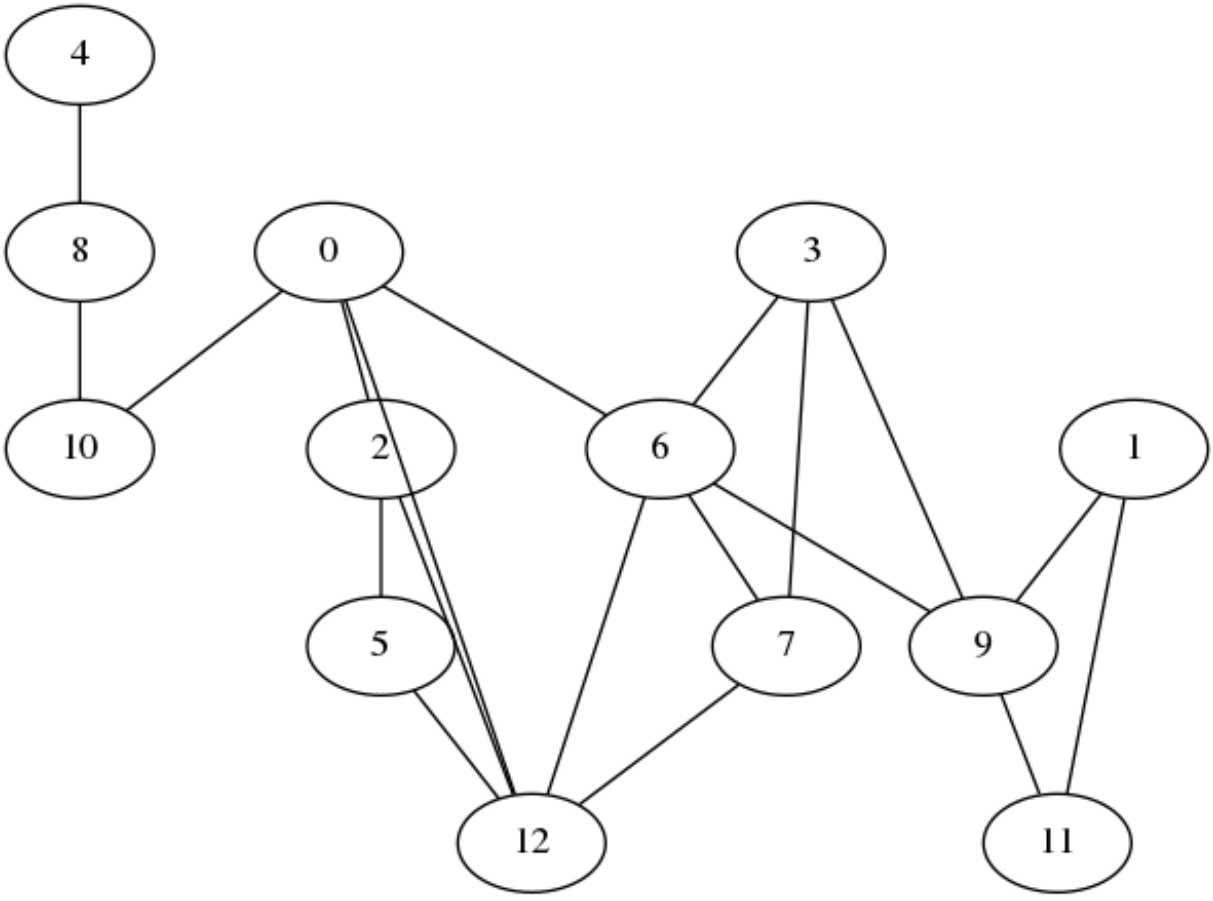


Рис. 1.2. Приклад топологічної схеми.

1.3. Маршрутизація в транспортних мережах

Маршрутизація – процес визначення в існуючій мережі одного або набору маршрутів, що є оптимальними відповідно до заданих характеристик пошуку, між заданими початковою та кінцевою мережевими вершинами [3]. Шляхом (маршрутом) називають організовану послідовність вузлів мережі та трактів передачі, що з'єднують між собою задані пари вузлів мережі. Приклад зображений на рис. 1.2.

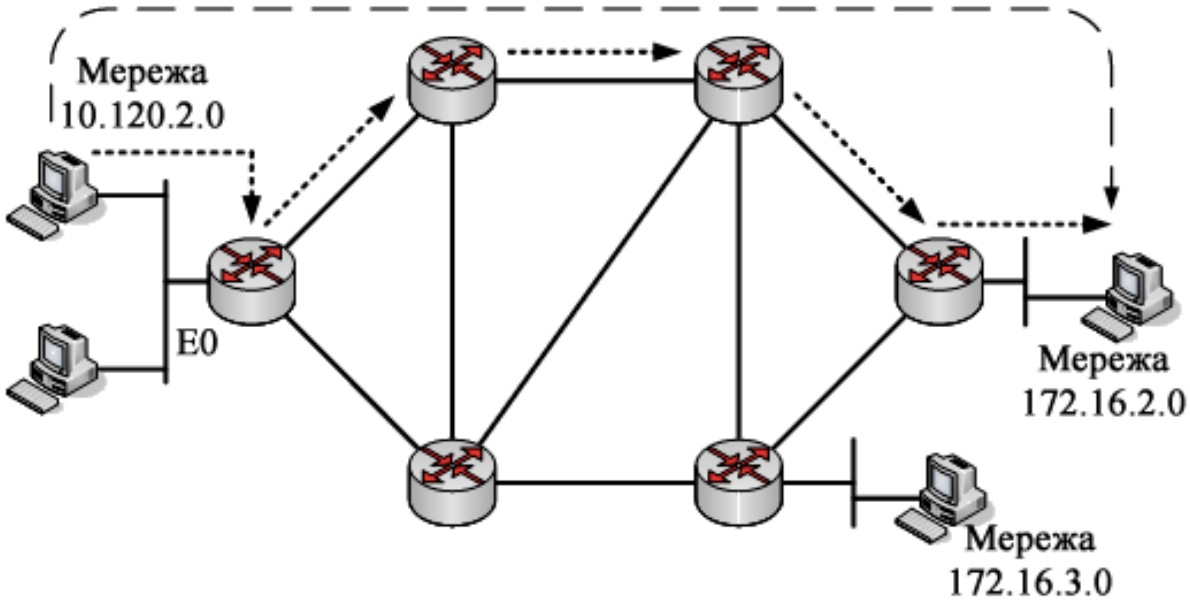


Рис. 1.2. Приклад маршруту в мережі

Для організації маршрутизації в мережах основною ціллю є збільшення показників якості обслуговування, приведення заданих показників до максимального значення. Ще однією із основних вимог є мінімізація характеристик передачі. Необхідно мінімізувати затримки в мережі та мінімізувати довжину кінцевих маршрутів, що дозволить більш швидко та надійно здійснювати передачу даних всередині заданої мережі. Наступною із не менш важливих вимог є забезпечення можливості збору та аналізу стану мережі. Стан мережі розглядається за наступними характеристиками:

- топологія;
- завантаження мережних ресурсів;

- затримки у каналах передачі;
- розрахунок шуканих маршрутів;
- реалізація маршрутних рішень.

Основою будь-якого протоколу маршрутизації є алгоритм маршрутизації, що буде застосовуватись для пошуку необхідних маршрутів доставки пакетів кінцевих даних. Вибір найоптимальнішого маршруту відбувається за допомогою алгоритму маршрутизації. Алгоритм маршрутизації – послідовність конкретних дій для визначення найкращого маршруту (шляху) для передачі пакетів кінцевих даних. Необхідно розглянути класифікацію алгоритмів маршрутизації. Алгоритми маршрутизації поділяються на групи по різним категоріям.

За показником оновлення можна виділити наступні алгоритми:

- статичні алгоритми – базуються на принципі ручного налаштування таблиць маршрутизації. Дані таблиці необхідно заповнити адміністратору до початку маршрутизації. Видалення доданих маршрутів відбувається також безпосередньо адміністратором ручним способом, як і зміна таких маршрутів. Недоліком таких алгоритмів є безпосередньо ручна їх організація та статичність. Застосування таких алгоритмів здійснюється лише в невеликих мережах з нескладною топологією заданої мережі.
- багатомаршрутні алгоритми – базуються на принципі динамічної зміни маршрутів відповідно до організації мережі. Зміна маршрутів змінюється відповідно до зміни топології. Перевагами такої організації алгоритму є можливість автоматичної реконфігурації в реальному часі, без ручного втручання. Недоліком такого алгоритму є складність реалізації.

За кількістю використання наявних маршрутів виділяють наступні алгоритми:

- прості одномаршрутні алгоритми – базуються на використанні лише одного маршруту для передачі пакетів кінцевих даних;

- складні багатомаршрутні алгоритми – базуються на використанні різних маршрутів з зазначеними пріоритетами. Присвоєння пріоритету відбувається за оцінкою завантаженості кожного з наявних маршрутів. Передача пакетів кінцевих даних відбувається лише по маршрутам з найвищим зазначеним пріоритетом. Маршрут з найвищим пріоритетом вважається найголовнішим для передачі даних. Решта маршрутів можуть бути використані як додаткові (резервні маршрути). Перевагою таких алгоритмів є можливість передачі даних по багатьом маршрутах. Таку передачу ще називають мультиплексною.

За структурою маршрутизації можна виділити наступні алгоритми:

- алгоритми з однорідною системою маршрутизації – базуються на принципі рівноправного доступу маршрутизаторів один до одного;
- алгоритми з ієрархічною системою маршрутизації – базуються на принципі ієрархічного доступу маршрутизаторів один до одного. Для ієрархічної системи маршрутизації відносяться два типи маршрутизаторів – базові та небазові. Пакети даних переміщаються від небазових маршрутизаторі до базових, до того моменту поки не досягнуть мережі призначення. Після досягнення мережі призначення пакети кінцевих даних переміщаються від останнього з базових маршрутизаторів до кінцевого пункту призначення, при цьому дані можуть проходити і через небазові маршрутизатори. Для даної системи маршрутизації відноситься встановлення груп вузлів, що отримали назву домени. Для доменів притаманна наявність двох типів маршрутизаторів – внутрішніх та зовнішніх. Першим притаманна можливість доступу тільки до маршрутизаторів лише в межах одного домену, зовнішні ж в свою чергу є складнішими маршрутизаторами та дозволяють виконувати міждоменну передачу даних. Зовнішні маршрутизатори мають складніші алгоритми маршрутизації, оскільки повинні забезпечувати міждоменну передачу даних, коли внутрішнім маршрутизаторам достатньо мати простий алгоритм для внутрішньої маршрутизації.

Цікавою особливістю є те, що задачу маршрутизації вирішують не тільки маршрутизатори а й крайові вузли. Відповідно до розглянутої особливості виділяють ще два типи алгоритмів:

- алгоритми з присутнім інтелектом в крайовому вузлі – базуються на принципі, що дозволяє визначити кінцевий маршрут передачі кінцевих даних, дана можливість забезпечується за рахунок наявного інтелектуального модуля в крайовій вершині. В таких системах відбувається ретрансляція пакетів кінцевих даних, що дозволяє забезпечити пересилку і зберігання пакетів. Такі системи маршрутизації найчастіше притаманні найкращим маршрутизаторам, однак недоліком таких систем є потреба в широкому пошуку та значній витраті часу на пошук трафіку;
- алгоритми з присутнім інтелектом всередині маршрутизатора – базуються на принципі маршрутизації безпосередньо за допомогою внутрішнього модуля самого маршрутизатора.

Ще однією із класифікацій є класифікація по способу обміну маршрутної інформації. До таких алгоритмів відносять два наступних типи:

- дистанційно-векторні алгоритми маршрутизації – базуються на принципі пересилки таблиці інформації або її частини всім своїм сусідам. Такі алгоритми ще мають назву алгоритми Беллмана-Форда;
- алгоритми стану каналу – базуються на принципі пересилки всієї необхідної інформації маршрутної таблиці, що описує стан власних каналів маршрутизатора, всім вузлам організованої, об’єднаної мережі. Такі алгоритми ще називають алгоритмами першочерговості найкоротшого маршруту. Перевагами таких алгоритмів є низька ймовірність організації петель маршрутизації, що присутня в дистанційно-векторних алгоритмах маршрутизації. Недоліками таких алгоритмів є велика складність реалізації та великі потреби в обчислювальних можливостях, що тягнуть за собою наявність великої потужності процесора та значної кількості пам’яті.

Розвиток сучасних транспортних мереж супроводжується інтенсифікацією різних транспортних потоків, що в свою чергу потребує розробки методів підвищення та покращення швидкодії маршрутів в мережі.

Маршрутизація в транспортних мережах – спосіб вибору найоптимальнішого маршруту серед мережі доріг, водних або ж повітряних шляхів з метою зменшення часу та вартості переміщення транспортного засобу по цьому шляху.

Вибір кінцевого маршруту може залежати від різних факторів:

- кількості, типу та технічного стану транспортного складу;
- наявності та експлуатаційних властивостей різних транспортних шляхів та мереж;
- значення затрат на перевезення, пов’язаних з конкретним маршрутом.

1.4. Порівняння алгоритмів аналогів

Існує декілька основних алгоритмів аналогів, що дозволяють виконувати маршрутизацію [4]. До таких алгоритмів відносяться комбінаторні алгоритми, такі алгоритми ще називають алгоритмами спрямованого перебору. Дані алгоритми дають можливість виконувати пошук найкоротших та найоптимальніших маршрутів. Безсумнівною перевагою таких алгоритмів є безпосередня проста та заздалегідь відома обчислювальна складність.

Серед таких алгоритмів найкращими та найпоширенішими вважаються наступні алгоритми:

- алгоритм Дійкстри ;
- алгоритм Беллмана-Форда;
- алгоритм Флойда-Уоршела.

Відмінність двох перших алгоритмів від останнього полягає в можливості перших двох алгоритмів знаходити найкоротші та найоптимальніші маршрути для доставки пакетів кінцевих даних, між початковою та кінцевою вершинами.

Алгоритм Флойда-Уоршела, що був наведений третім – відрізняється від двох попередніх, оскільки його особливістю є пошук всіх можливих шляхів для доставки пакетів кінцевих даних між всіма вузлами.

Слід розглянути кожен із даних алгоритмів детальніше аби детально розібрати усі деталі а також переваги та недоліки. Для демонстрації роботи алгоритмів використано спеціальну структуру мережі, що продемонстрована на рис. 1.4.

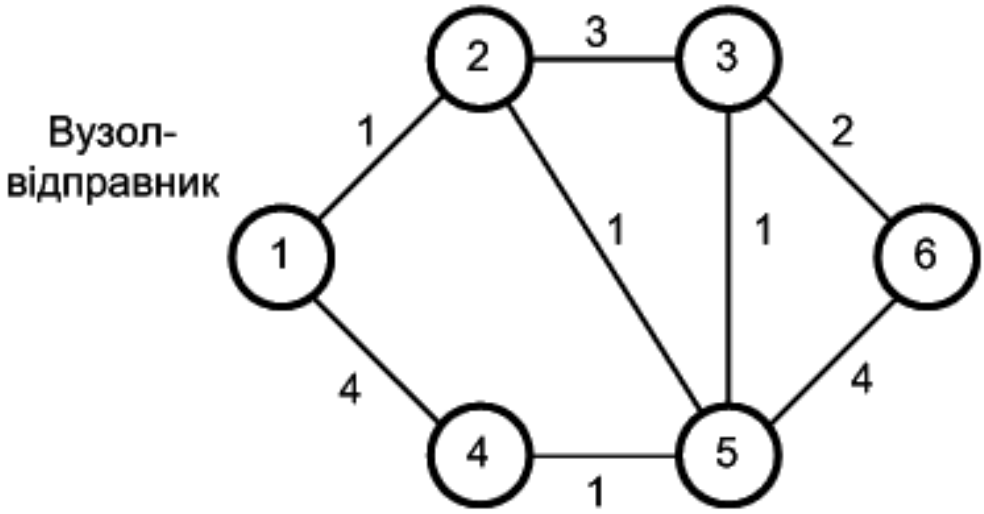


Рис. 1.4. Приклад структури мережі

Першим слід розглянути алгоритм Дійкстри. Даному алгоритму притаманна особливість, що полягає у вимогах до топологої. Значення усіх ребер вибраної топології завжди повинні набувати позитивного значення. Дана вимога, як правило реалізована в топологіях сучасних мереж. Основною ідеєю даного алгоритму є пошук найкоротших маршрутів в порядку зростання їх зазначеної ваги. Найкоротшим шляхом вважається маршрут що складається лише з однієї ланки, такий маршрут можна охарактеризувати як будь-який шлях від вершини до її сусідньої. Якщо ж таких шляхів, що складаються лише з однієї ланки декілька, то обирається такий, вага ланки якого є найменшою. Наступним найкоротшим шляхом будемо вважати маршрут, що складається також лише з однієї ланки та має більше значення ваги ніж попередній, або

такий що складається з двох ланок, однак одна із його ланок належить до попереднього маршруту. Що до складності алгоритму Дійкстри, то оскільки число усіх можливих операцій є пропорційним M, а самі кроки повторюється з інтервалом в (M-1), то обсяг усіх обчислень при найгіршому розкладі становить $O(m^2)$. Даний алгоритм відноситься до алгоритмів стану каналу.

Приклад роботи алгоритму Дійкстри продемонстрований на рис. 1.5.

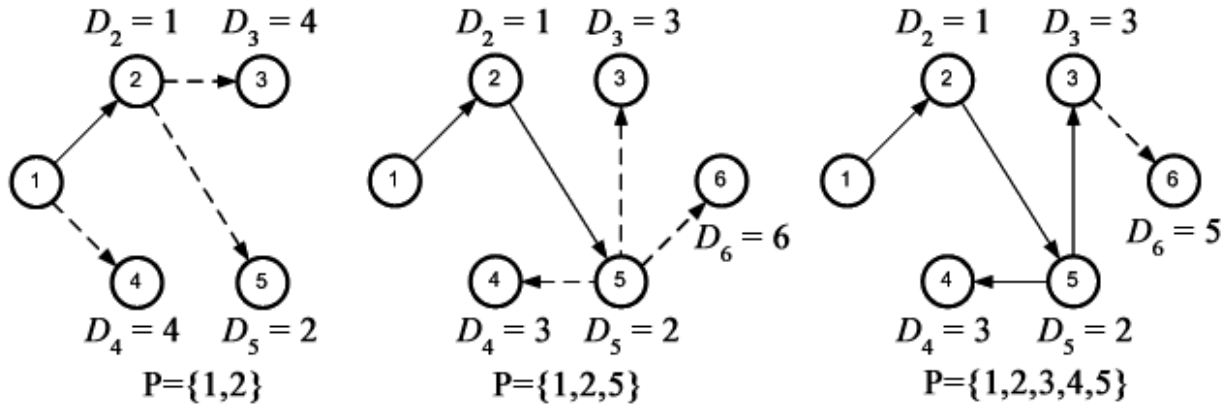


Рис. 1.5. Приклад роботи алгоритму Дійкстри

Другим слід розглянути алгоритм Беллмана-Форда. Даний алгоритм був названий в честь двох вчених Беллмана та Форда, що вперше опублікували ідею даного алгоритму. Ідея даного алгоритму базується на принципі збереження в таблиці списку усіх відомих та можливих маршрутів. Додатково виконується помітка в кожному з елементів таблиці інформація про кінцевого одержувача та кількість переходів мережі. Ще однією із особливостей є здійснення пересилання копії створеної таблиці усім іншим маршрутизаторам, до яких в маршрутизатора, що здійснює надсилання, є безпосередній доступ. Після надсилання копії таблиці першин маршрутизатором (в подальшому маршрутизатор 1) маршрутизатору під номером 2, маршрутизатор під номером 2 отримає надіслану копію таблиці та виконає аналіз переданої інформації. До такої інформації слід віднести відстані між адресатами та сама інформація про адресатів. Наступним кроком є виконання заміни шляхів відповідно до інформації з копії таблиці. Така заміна може бути виконана,

якщо в отриманій копії таблиці містяться маршрути, що є коротшими або оптимальнішими за маршрути, які доступні маршрутизатору під номером 2. Також виконується додавання шляхів, якщо таких в таблиці маршрутизатора не було виявлено. Розглянута таблиця маршрутизація використовується наведеним алгоритмом для пошуку метрики маршруту, також на основі таблиці виконується пошук мінімального числа всіх можливих пересилань.

Що до складності даного алгоритму, то варто порахувати складність на основі ітерацій. Для найгіршого варіанту кількість ітерацій буде рівна (M-1), в той же час кожна з ітерацій має бути виконана для (M-1) вершини. В підсумку, складність для найгіршого проходу виконання алгоритму буде рівним $O(m^3)$. Однак існує і більш точно визначена формула складності виконання, вона рівна значенню O(a*n). Де a – значення кількості дуг, а n – значення максимальної кількості дуг, що присутня в найкоротшому маршруті.

Приклад роботи алгоритму Беллмана-Форда продемонстрований на рис. 1.6.

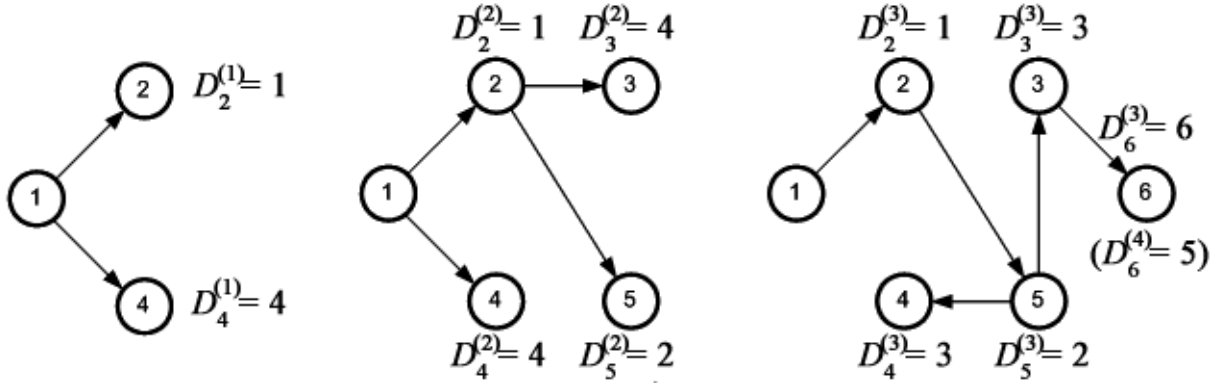


Рис. 1.6. Приклад роботи алгоритму Беллмана-Форда

Останнім слід розглянути алгоритм Флойда-Уоршела. Даний алгоритм також був названий в честь одного із вчених. Відмінністю даного алгоритму являється пошук не одного найкоротшого маршруту, а пошук відразу всіх шляхів для всіх заданих пар вузлів. Число ітерацій для даного алгоритму напряму залежить від кількості усіх вершин мережі. Особливістю даного алгоритму є можливість використання проміжних вершин на маршрутах,

пошук яких здійснюється. Даний алгоритм має схожі принципи пошуку маршрутів, як і два попередньо наведені алгоритми. Дана подібність полягає у пошуку шляхів, які на початку пошуку складаються лише з однієї ланки, не включаючи в свій шлях проміжні шляхи. Такі маршрути, що складаються лише з однієї ланки, слугують як критерій для пошуку найкоротших та найоптимальніших маршрутів. Наступним кроком є пошук найкоротших шляхів з використанням обмеженої кількості проміжних вершин. Для початку проміжною вершиною може бути тільки вершина 1, на наступному кроці в обмеження проміжних вершин будуть належати вершини 1 та 2. Така послідовність кроків та обмежень буде виконуватись до моменту коли всі шляхи будуть знайдені.

Складність наведеного алгоритму складає $O(m^3)$. Дана складність дорівнює такому значенню, за рахунок того, що кількість кроків алгоритму повторюється для кожної пари вузлів.

ВИСНОВКИ ДО РОЗДІЛУ 1

В даному розділі було розглянуто та проаналізовано, що транспортні мережі є одним із видів існуючих мереж, що дозволяють організувати рух транспорту та спростити міську та міжміську комунікацію.

Було виявлено, що для можливості реалізації транспортної мережі існують спеціальні способи моделювання. Такі способи були спеціально розроблені для ефективної та простої побудови транспортних мереж, а також для можливості їх зручної організації. Методи моделювання транспортних мереж в деталях дозволяють реалізувати та організувати роботу транспортних мереж в цілому.

Також було детально розглянуто методи маршрутизації в транспортних мережах та проаналізовано всі можливі способи маршрутизації. Відповідно після детально проведеного аналізу було розроблену спеціальну класифікацію всіх можливих методів маршрутизації за різними ознаками. Розроблена класифікація дозволила просто та ефективно оцінити всі переваги та недоліки кожного з методів маршрутизації.

У останньому підрозділі було розглянуто всі можливі аналоги алгоритмів транспортної маршрутизації. Згідно із усіма описаними та розглянутими алгоритмами маршрутизації було розроблено відповідну класифікацію, що в свою чергу дозволила детальніше оцінити усі переваги та недоліки наведених алгоритмів. Також було розглянуто приклади роботи кожного з проаналізованих алгоритмів та проведено оцінку складності цих алгоритмів, що дало можливість виконати оцінку затрат їх програмної реалізації.

РОЗДІЛ 2
АЛГОРИТМ МАРШРУТИЗАЦІЇ

2.1. Опис основних етапів алгоритму

Даний алгоритм є таким, що обробляє всю інформацію почергово, але з врахуванням деяких деталей для майбутніх обчислень. Обробка даних для обчислення на майбутніх ітераціях є безсумнівною перевагою та особливістю даного алгоритму, так як основні обчислення алгоритму розвантажуються на рівномірні частини та дозволяють прискорити виконання пошуку оптимальних шляхів для передачі даних.

Можна виокремити основні етапи алгоритму:

1) Перевірка структури на наявність в ній вже знайдених шляхів між шуканими, в даний момент часу, вершинами. Даний етап дозволяє оптимізувати пошук найоптимальніших шляхів за рахунок лінійного пошуку структури з наразі знайденими шляхами.
2) Пошук всіх можливих шляхів між двома вершинами. Даний етап дозволяє знайти всі можливі шляхи між двома заданими вершинами, та безпосередньо зберегти їх до спеціальної структури, аби на наступних ітераціях мати можливість не виконувати пошук в ширину та шукати дані шляхи знову, а вибрати вже знайдені шляхи з структури та продовжити наступні кроки алгоритму для визначення найоптимальнішого шляху в даний момент часу. Дане рішення суттєво прискорює роботу алгоритму.
3) Перевірка знайдених шляхів. Даний етап дозволяє перевірити знайдені або вибрані із структури шляхи на можливість використання їх як оптимальних.
4) Пошук серед перевірених шляхів найоптимальнішого. Даний етап дозволяє вибрати найоптимальніший шлях, аналізуючи пропускну здатність кожного з перевірених шляхів на попередньому етапі.

5) Зменшення пропускної здатності для всіх ділянок знайденого шляху. Даний етап дозволяє зменшити пропускну здатність для всіх ділянок шляху, що в свою чергу свідчить про те, що оптимальність цього часу зменшується на майбутніх ітераціях.

2.2. Формування множини шляхів

Важливим фактором для алгоритмів на графах є часова та алгоритмічна складність, оскільки пошук існуючих шляхів повинен працювати найшвидшим чином та завжди за константний час. Тому відповідно для швидкого пошуку всіх можливих шляхів між двома вершинами було розроблено спеціальний алгоритм, що базується на алгоритмі пошуку у ширину (BFS – Breadth First Search) [5]. Реалізація даного алгоритму дозволяє рекурсивно знайти всі можливі шляхи між двома заданими вершинами за константний час [6]. Часова складність алгоритму становить N*O(V+E). Де N=E і означає кількість вершин, а E означає кількість ребер на графі [7]. Даний алгоритм є лише основою основного алгоритму, який в свою чергу дозволяє не просто знайти всі можливі шляхи, а знайти найоптимальніший шлях між двома вершинами в конкретний момент часу. Вимоги до такого алгоритму є високими, особливо до його оптимізацій та часу виконання. Для знаходження найоптимальнішого шляху на графі в конкретний момент часу було розроблено спеціальний алгоритм, в якому за основу було взято алгоритм пошуку в ширину. Особливістю розробленого алгоритму є можливість оптимізації пошуку шляхів поміж двох вершин. Дана особливість полягає у можливості не виконувати пошук повторно, якщо шляхи були вже знайдені для певних вершин. Таким чином після пошуку всіх можливих шляхів ці шляхи зберігаються в спеціальну структуру, при наступному запиті на пошук найоптимальнішого шляху, пошук шляхів відбуватись не буде, якщо структура вже містить знайдені всі можливі вершини для заданих вершин.

2.3. Приклад роботи алгоритму

Розглянемо приклад модифікованого пошуку в ширину, що формує всі можливі шляхи між двома вершинами, взявши для розгляду граф, зображений на рис. 2.1. Всі можливі шляхи будемо формувати між вершинами V_1 та V_{12}.

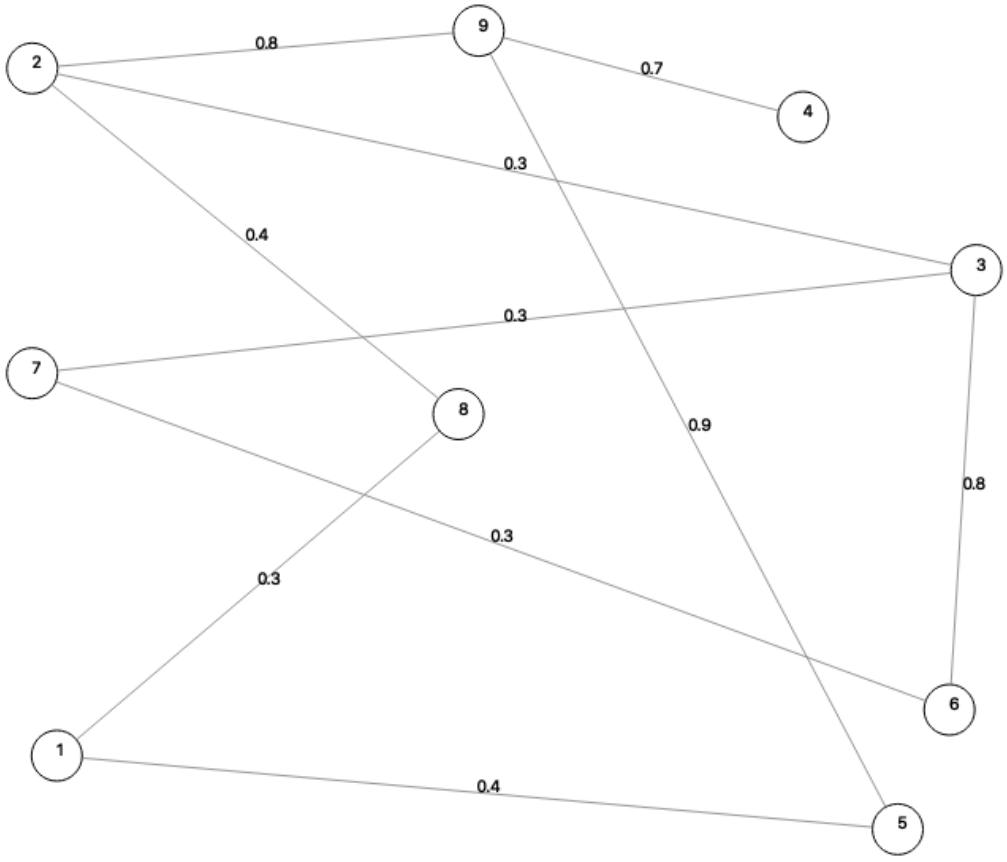


Рис. 2.1. Граф мережі

Всі можливі шляхи між вершинами V_s та V_d позначимо жирними лініями. Першим кроком будемо вважати перевірку спеціальної структури на наявність в ній вже знайдених маршрутів. Якщо структура вже містить знайдені маршрути між двома заданими вершинами, то відразу переходимо до кроку під номером 3. Однак оскільки ми виконуємо тільки перший пошук найоптимальнішого шляху, то відповідно наша спеціальна структура ще не містить жодних раніше знайдених шляхів.

Другим кроком будемо вважати пошук всіх можливих шляхів між вершинами V_s та V_d, де V_s – початкова вершина (Source Vertex), V_d – кінцева вершина (Destination Vertex).

Для прикладу розглянемо пошук найоптимальнішого маршруту між вершинами V_1 та V_7.

На першому кроці починаємо побудову множини всіх можливих шляхів за допомогою рекурсивного пошуку в ширину (Recursive BFS – Recursive Breadth-First Search), що дозволить нам знайти всі можливі варіанти шляхів між заданими вершинами, в результаті пошуку ми отримаємо наступну множину всіх можливих шляхів:

$P_1 = \{ V_1 - V_5 - V_9 - V_2 - V_3 - V_6 - V_7 \}$;

$P_2 = \{ V_1 - V_5 - V_9 - V_2 - V_3 - V_7 \}$;

$P_3 = \{ V_1 - V_8 - V_2 - V_3 - V_6 - V_7 \}$;

$P_4 = \{ V_1 - V_8 - V_2 - V_3 - V_7 \}$.

Третім кроком необхідно перевірити та помітити всі непридатні шляхи серед знайдених шляхів $P_{1\text{-}4}$, якщо такі присутні. Непридатним шляхом будемо вважати такі, в яких принаймні хоча б на одній ділянці маршруту пропускна здатність не перевищує 0.1. Максимальною пропускною здатністю будемо вважати 1.0. Еквівалентними до числових значень можна вважати відсоткові відношення, відповідно до наведених значень будемо вважати 0.1 – 10% та 1 – 100%. Відповідно до цього розглянемо пропускну здатність для кожної ділянки на кожному з наведених маршрутів у вигляді матриці та помітимо всі непридатні маршрути, якщо такі наявні.

Таблиця 2.1

	V_1	V_2	V_3	V_5	V_6	V_7	V_8	V_9
V_1	0	0	0	0.4	0	0	0.3	0
V_2	0	0	0.3	0	0	0	0	0
V_3	0	0	0	0	0.8	0.3	0	0
V_5	0	0	0	0	0	0	0	0.9

Продовження Таблиці 2.1

	V_1	V_2	V_3	V_5	V_6	V_7	V_8	V_9
V_6	0	0	0	0	0	0.3	0	0
V_7	0	0	0	0	0	0	0	0
V_8	0	0.4	0	0	0	0	0	0
V_9	0	0.8	0	0	0	0	0	0

Дана матриця пропускних здатностей не є повною, оскільки в неї входять пропускні здатності тільки для знайдених маршрутів на кроці під номером 2.

Головна діагональ матриці заповнена нулями, оскільки на графі немає петель, решта ж значень матриці дорівнює 0 тому що деякі з переходів не є ділянками в наведених шляхах, а також тому що матриця не є симетричною, оскільки немає зворотних шляхів до знайдених.

Відповідно до матриці пропускні здатності для кожної з ділянок можемо відобразити у більш наглядному вигляді. Для позначення пропускних здатностей введемо позначення $C_{i,j}$ – де C – пропускна здатність між двома вершинами, а i та j – відповідно номери вершин поміж яких дана пропускна здатність задана. Пропускними здатностями для всього шляху будемо вважати позначення – P_{kC}, де k – номер кожного із знайдених маршрутів.

1) Пропускні здатності для маршруту P_1:

   $P_{1C} = \{ C_{1,5} – C_{5,9} – C_{9,2} – C_{2,3} – C_{3,6} – C_{6,7} \}$;

   $C_{1,5} = 0.4$;

   $C_{5,9} = 0.9$;

   $C_{9,2} = 0.8$;

   $C_{2,3} = 0.3$;

   $C_{3,6} = 0.8$;

   $C_{6,7} = 0.3$;

   $P_{1C} = \{ 0.4 – 0.9 – 0.8 – 0.3 – 0.8 – 0.3 \}$.

2) Пропускні здатності для маршруту P_2:

$P_{2C} = \{ C_{1,5} – C_{5,9} – C_{9,2} – C_{2,3} – C_{3,7} \};$

$C_{1,5} = 0.4;$

$C_{5,9} = 0.9;$

$C_{9,2} = 0.8;$

$C_{2,3} = 0.3;$

$C_{3,7} = 0.3;$

$P_{2C} = \{ 0.4 – 0.9 – 0.8 – 0.3 – 0.3 \}.$

3) Пропускні здатності для маршруту P_3:

$P_{3C} = \{ C_{1,8} – C_{8,2} – C_{2,3} – C_{3,6} – C_{6,7} \};$

$C_{1,8} = 0.3;$

$C_{8,2} = 0.4;$

$C_{2,3} = 0.3;$

$C_{3,6} = 0.8;$

$C_{6,7} = 0.3;$

$P_{3C} = \{ 0.3 – 0.4 – 0.3 – 0.8 – 0.3 \}.$

4) Пропускні здатності для маршруту P_4:

$P_{4C} = \{ C_{1,8} – C_{8,2} – C_{2,3} – C_{3,7} \};$

$C_{1,8} = 0.3;$

$C_{8,2} = 0.4;$

$C_{2,3} = 0.3;$

$C_{3,7} = 0.3;$

$P_{4C} = \{ 0.3 – 0.4 – 0.3 – 0.3 \}.$

Відповідно до перевірених пропускних здатностей на кожній з ділянок кожного з маршрутів можна впевнитись, що непридатних шляхів немає і тому всі маршрути можуть бути розглянуті, як оптимальні.

Четвертим кроком будемо вважати пошук найоптимальнішого шляху серед всіх наявних. Для цього скористаємось формулою та перевіримо кожен з непомічених на непридатність. Для обрахунку найоптимальнішого шляху використаємо наступну формулу:

$$A_{Pk} = \sum_{m=1}^{n} (1 - C_{i,j});\ \overline{i,j = 1,V};$$

Як було розглянуто раніше $C_{i,j}$ – пропускна здатність конкретної ділянки певного маршруту з наявних, а індекси і та j відповідно індекси вершин.

Скориставшись даною формулою обрахуємо всі чисельні значення для кожного з доступних маршрутів:

1) Для маршруту P_1 та пропускних здатностей для цього маршруту P_{1C}:

$$A_{P1} = \sum(1 - C_{1,5}) + (1 - C_{5,9}) + (1 - C_{9,2}) + (1 - C_{2,3}) + (1 - C_{3,6}) + (1 - C_{6,7}) = \sum(1 - 0.4) + (1 - 0.9) + (1 - 0.8) + (1 - 0.3) + (1 - 0.8) + (1 - 0.3) = 2.5;$$

2) Для маршруту P_2 та пропускних здатностей для цього маршруту P_{2C}:

$$A_{P2} = \sum(1 - C_{1,5}) + (1 - C_{5,9}) + (1 - C_{9,2}) + (1 - C_{2,3}) + (1 - C_{3,7}) = \sum(1 - 0.4) + (1 - 0.9) + (1 - 0.8) + (1 - 0.3) + (1 - 0.3) = 2.3;$$

3) Для маршруту P_3 та пропускних здатностей для цього маршруту P_{3C}:

$$A_{P3} = \sum(1 - C_{1,8}) + (1 - C_{8,2}) + (1 - C_{2,3}) + (1 - C_{3,6}) + (1 - C_{6,7}) = \sum(1 - 0.3) + (1 - 0.4) + (1 - 0.3) + (1 - 0.8) + (1 - 0.3) = 2.9;$$

4) Для маршруту P_4 та пропускних здатностей для цього маршруту P_{4C}:

$$A_{P4} = \sum(1 - C_{1,8}) + (1 - C_{8,2}) + (1 - C_{2,3}) + (1 - C_{3,7}) = \sum(1 - 0.3) + (1 - 0.4) + (1 - 0.3) + (1 - 0.3) = 2.7;$$

Згідно з розрахованих значень для кожного з маршрутів необхідно визначити мінімальне серед вище знайдених, що дасть нам змогу визначити

який із шляхів є найоптимальніший, що в свою чергу супроводжується наявністю найвищих значень пропускних здатностей на кожній ділянці серед усіх придатних маршрутів.

Мінімальним серед порахованих значень є значення $A_{P2} = 2.3$. Відповідно до цього в даний момент моменту часу найоптимальнішим шляхом є маршрут P_2 з пропускними здатностями $P_{2C} = \{ C_{1,5} – C_{5,9} – C_{9,2} – C_{2,3} – C_{3,7} \}$ та вершинами $P_2 = \{ V_1 - V_5 - V_9 - V_2 - V_3 - V_6 \}$. Даний маршрут вважається найоптимальнішим за рахунок найбільшої кількості ділянок з найвищою пропускною здатністю серед інших, а також в через те, що він є одним із найкоротших серед інших.

П’ятим кроком будемо вважати зменшення пропускної здатності на кожній з ділянок на знайденому маршруті на 0.1 – еквівалент 10%. Для забезпечення зменшення пропускної здатності на кожній з ділянок всього маршруту необхідно по черзі зменшити пропускну здатність кожної з ділянок. В результаті отримаємо для маршруту P_2 наступне:

$P_{2C} = \{ C_{1,5} – C_{5,9} – C_{9,2} – C_{2,3} – C_{3,7} \}$;

$C_{1,5} = 0.3$;

$C_{5,9} = 0.8$;

$C_{9,2} = 0.7$;

$C_{2,3} = 0.2$;

$C_{3,7} = 0.2$;

$P_{2C} = \{ 0.3 – 0.8 – 0.7 – 0.2 – 0.2 \}$.

Відповідно до цього матриця пропускних здатностей також набуде іншого вигляду:

Таблиця 2.2

	V_1	V_2	V_3	V_5	V_6	V_7	V_8	V_9
V_1	0	0	0	0.3	0	0	0.3	0
V_2	0	0	0.2	0	0	0	0	0
V_3	0	0	0	0	0.8	0.2	0	0

Продовження Таблиці 2.2

	V_1	V_2	V_3	V_5	V_6	V_7	V_8	V_9
V_5	0	0	0	0	0	0	0	0.8
V_6	0	0	0	0	0	0.3	0	0
V_7	0	0	0	0	0	0	0	0
V_8	0	0.4	0	0	0	0	0	0
V_9	0	0.7	0	0	0	0	0	0

Відразу після зменшення пропускних здатностей на кожній ділянці знайденого маршруту ми отримуємо найоптимальніший маршрут P_2 між заданими двома вершинами. На графі продемонстрований найоптимальніший маршрут P_2, як результат роботи алгоритму. Маршрут P_2 виділений чорним жирним кольором на рис. 2.2.

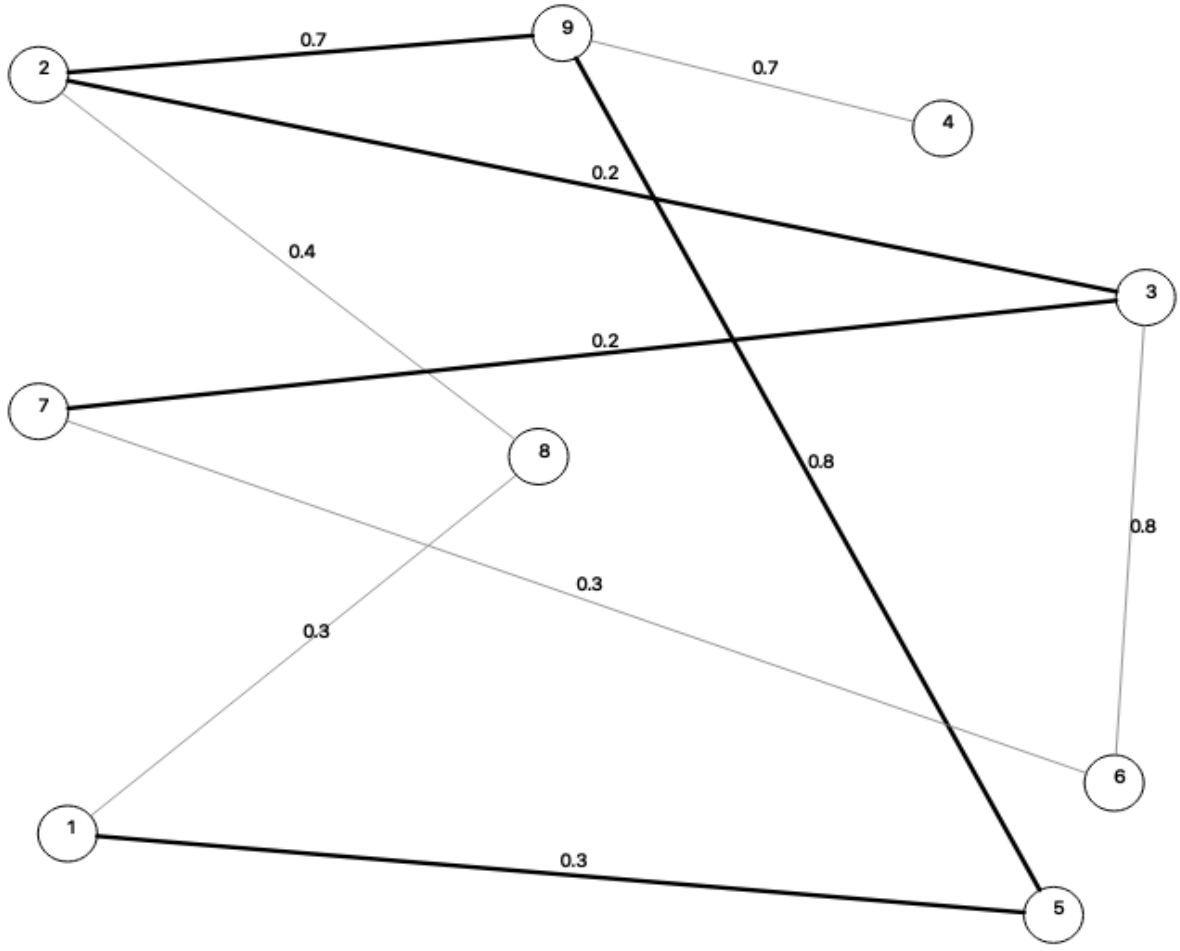


Рис. 2.2. Найоптимальніший маршрут P_2

Відповідно до усіх розглянутих кроків алгоритму необхідно розглянути роботу даного алгоритму, але вже при наявності знайдених маршрутів в спеціальній структурі, що демонструє певну оптимізацію при виконанні алгоритму.

Першим кроком також буде вважатись перевірка структури на наявність вже знайдених маршрутів між заданими вершинами. Заданими вершинами будемо вважати вершини V_1 та V_7. Ми обрали такі ж вершини, як і у попередньому прикладі аби продемонструвати роботу алгоритму без пошуку шляхів за допомогою рекурсивного пошуку в ширину, що в свою чергу нам дозволить використати вже знайдені шляхи у структурі та отримати швидший час виконання алгоритму. Після перевірки структури ми отримаємо наступну множину маршрутів:

$P_1 = \{ V_1 - V_5 - V_9 - V_2 - V_3 - V_6 - V_7 \};$

$P_2 = \{ V_1 - V_5 - V_9 - V_2 - V_3 - V_7 \};$

$P_3 = \{ V_1 - V_8 - V_2 - V_3 - V_6 - V_7 \};$

$P_4 = \{ V_1 - V_8 - V_2 - V_3 - V_7 \}.$

Другий крок виконуватись не буде, так як ми вже маємо знайдені маршрути між заданими вершинами, згідно з цим необхідно перейти до виконання кроку під номером 3.

Третім кроком, так як і в попередньому прикладі, необхідно перевірити та помітити всі непридатні шляхи серед знайдених шляхів $P_{1\text{-}4}$, якщо такі присутні. Для перевірки та маркування всіх непридатних шляхів скористаємось вже раніше створеною матрицею пропускних здатностей – Таблиця 2.2.

Відповідно до Таблиці 2.2 можемо скласти маршрути пропускних здатностей:

1) Пропускні здатності для маршруту P_1:

$P_{1C} = \{ C_{1,5} – C_{5,9} – C_{9,2} – C_{2,3} – C_{3,6} – C_{6,7} \};$

$C_{1,5} = 0.3$;

$C_{5,9} = 0.8$;

$C_{9,2} = 0.7$;

$C_{2,3} = 0.2$;

$C_{3,6} = 0.8$;

$C_{6,7} = 0.3$;

$P_{1C} = \{ 0.3 – 0.8 – 0.7 – 0.2 – 0.8 – 0.3 \}$.

2) Пропускні здатності для маршруту P_2:

$P_{2C} = \{ C_{1,5} – C_{5,9} – C_{9,2} – C_{2,3} – C_{3,7} \}$;

$C_{1,5} = 0.3$;

$C_{5,9} = 0.8$;

$C_{9,2} = 0.7$;

$C_{2,3} = 0.2$;

$C_{3,7} = 0.2$;

$P_{2C} = \{ 0.3 – 0.8 – 0.7 – 0.2 – 0.2 \}$.

3) Пропускні здатності для маршруту P_3:

$P_{3C} = \{ C_{1,8} – C_{8,2} – C_{2,3} – C_{3,6} – C_{6,7} \}$;

$C_{1,8} = 0.3$;

$C_{8,2} = 0.4$;

$C_{2,3} = 0.2$;

$C_{3,6} = 0.8$;

$C_{6,7} = 0.3$;

$P_{3C} = \{ 0.3 – 0.4 – 0.2 – 0.8 – 0.3 \}$.

4) Пропускні здатності для маршруту P_4:

$P_{4C} = \{ C_{1,8} – C_{8,2} – C_{2,3} – C_{3,7} \}$;

$C_{1,8} = 0.3$;

$C_{8,2} = 0.4$;

$C_{2,3} = 0.2$;

$C_{3,7} = 0.2$;

P_{4C} = { 0.3 – 0.4 – 0.2 – 0.2 }.

Відповідно до перевірених пропускних здатностей на кожній з ділянок кожного з маршрутів можна впевнитись, що непридатних шляхів немає і тому всі маршрути можуть бути розглянуті, як оптимальні та придатні.

Четвертим кроком є пошук найоптимальнішого шляху серед усіх придатних. Для цього скористаємось формулою та перевіримо кожен з непомічених на непридатність.

1) Для маршруту P_1 та пропускних здатностей для цього маршруту P_{1C}:

$$A_{P1} = \sum(1 - C_{1,5}) + (1 - C_{5,9}) + (1 - C_{9,2}) + (1 - C_{2,3}) + (1 - C_{3,6}) + (1 - C_{6,7}) = \sum(1 - 0.3) + (1 - 0.8) + (1 - 0.7) + (1 - 0.2) + (1 - 0.8) + (1 - 0.3) = 2.9;$$

2) Для маршруту P_2 та пропускних здатностей для цього маршруту P_{2C}:

$$A_{P2} = \sum(1 - C_{1,5}) + (1 - C_{5,9}) + (1 - C_{9,2}) + (1 - C_{2,3}) + (1 - C_{3,7}) = \sum(1 - 0.3) + (1 - 0.8) + (1 - 0.7) + (1 - 0.2) + (1 - 0.2) = 2.8;$$

3) Для маршруту P_3 та пропускних здатностей для цього маршруту P_{3C}:

$$A_{P3} = \sum(1 - C_{1,8}) + (1 - C_{8,2}) + (1 - C_{2,3}) + (1 - C_{3,6}) + (1 - C_{6,7}) = \sum(1 - 0.3) + (1 - 0.4) + (1 - 0.2) + (1 - 0.8) + (1 - 0.3) = 3.0;$$

4) Для маршруту P_4 та пропускних здатностей для цього маршруту P_{4C}:

$$A_{P4} = \sum(1 - C_{1,8}) + (1 - C_{8,2}) + (1 - C_{2,3}) + (1 - C_{3,7}) = \sum(1 - 0.3) + (1 - 0.4) + (1 - 0.2) + (1 - 0.2) = 2.9;$$

Згідно з розрахованих значень для кожного з маршрутів необхідно визначити мінімальне серед вище знайдених, що дасть нам змогу визначити

який із шляхів є найоптимальніший, що в свою чергу супроводжується наявністю найвищих значень пропускних здатностей на кожній ділянці серед усіх придатних маршрутів.

Мінімальним серед порахованих значень є значення $A_{P2} = 2.8$. Відповідно до цього в даний момент моменту часу найоптимальнішим шляхом є маршрут P_2 з пропускними здатностями $P_{2C} = \{ C_{1,5} – C_{5,9} – C_{9,2} – C_{2,3} – C_{3,7} \}$ та вершинами $P_2 = \{ V_1 - V_5 - V_9 - V_2 - V_3 - V_6 \}$. Даний маршрут вважається найоптимальнішим за рахунок найбільшої кількості ділянок з найвищою пропускною здатністю серед інших, а також в через те, що він є одним із найкоротших серед інших.

П’ятим кроком є зменшення пропускної здатності на кожній з ділянок на знайденому маршруті на 0.1 – еквівалент 10%. Для забезпечення зменшення пропускної здатності на кожній з ділянок всього маршруту необхідно по черзі зменшити пропускну здатність кожної з ділянок. В результаті отримаємо для маршруту P_2 наступне:

$P_{2C} = \{ C_{1,5} – C_{5,9} – C_{9,2} – C_{2,3} – C_{3,7} \}$;

$C_{1,5} = 0.2$;

$C_{5,9} = 0.7$;

$C_{9,2} = 0.6$;

$C_{2,3} = 0.1$;

$C_{3,7} = 0.1$;

$P_{2C} = \{ 0.2 – 0.7 – 0.6 – 0.1 – 0.1 \}$.

Відповідно до цього матриця пропускних здатностей також набуде іншого вигляду:

Таблиця 2.3

	V_1	V_2	V_3	V_5	V_6	V_7	V_8	V_9
V_1	0	0	0	0.2	0	0	0.3	0
V_2	0	0	0.1	0	0	0	0	0
V_3	0	0	0	0	0.8	0.1	0	0

Продовження Таблиці 2.3

	V_1	V_2	V_3	V_5	V_6	V_7	V_8	V_9
V_5	0	0	0	0	0	0	0	0.7
V_6	0	0	0	0	0	0.3	0	0
V_7	0	0	0	0	0	0	0	0
V_8	0	0.4	0	0	0	0	0	0
V_9	0	0.6	0	0	0	0	0	0

Відразу після зменшення пропускних здатностей на кожній ділянці знайденого маршруту ми отримуємо найоптимальніший маршрут P_2 між заданими двома вершинами. На графі продемонстрований найоптимальніший маршрут P_2, як результат роботи алгоритму. Маршрут P_2 виділений чорним жирним кольором на рис. 2.3.

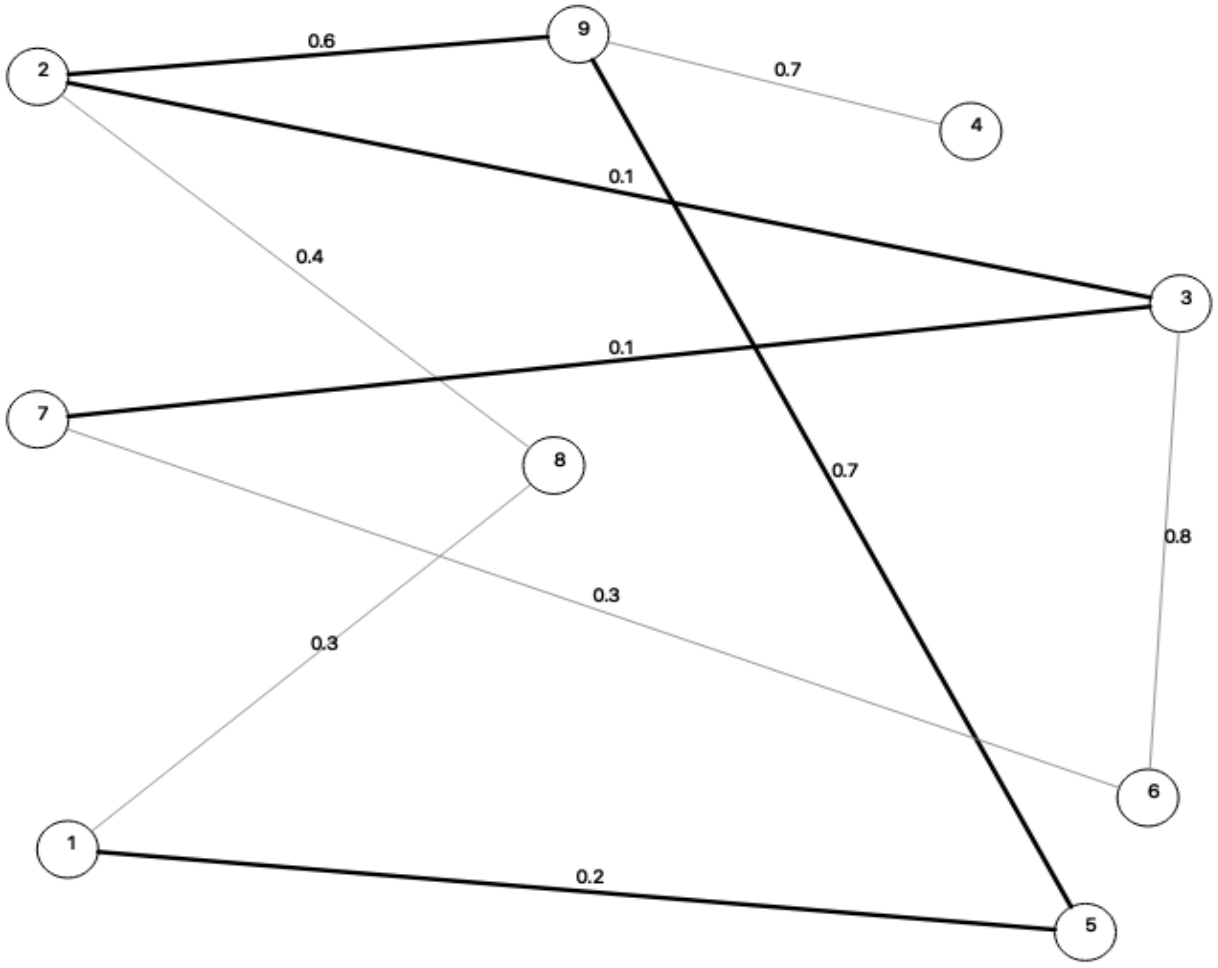


Рис. 2.3. Найоптимальніший маршрут P_2

Якщо ми виконаємо пошук найоптимальнішого шляху з заданими вершинами V_1 та V_7 знову, то ми не отримаємо бажаного результату, оскільки всі шляхи на кроці алгоритму під номером 3 будуть непридатними так як кожен з всіх наявних шляхів в структурі містить принаймні одну ділянку з пропускною здатністю в 10%. Відповідно до Таблиці 2.3 перевіримо, що кожен з наявних маршрутів є непридатним:

1) Пропускні здатності для маршруту P_1:

   $P_{1C} = \{ C_{1,5} – C_{5,9} – C_{9,2} – C_{2,3} – C_{3,6} – C_{6,7} \};$

   $C_{1,5} = 0.2;$

   $C_{5,9} = 0.7;$

   $C_{9,2} = 0.6;$

   $C_{2,3} = 0.1;$

   $C_{3,6} = 0.8;$

   $C_{6,7} = 0.3;$

   $P_{1C} = \{ 0.2 – 0.7 – 0.6 – 0.1 – 0.8 – 0.3 \}.$

2) Пропускні здатності для маршруту P_2:

   $P_{2C} = \{ C_{1,5} – C_{5,9} – C_{9,2} – C_{2,3} – C_{3,7} \};$

   $C_{1,5} = 0.2;$

   $C_{5,9} = 0.7;$

   $C_{9,2} = 0.6;$

   $C_{2,3} = 0.1;$

   $C_{3,7} = 0.1;$

   $P_{2C} = \{ 0.2 – 0.7 – 0.6 – 0.1 – 0.1 \}.$

3) Пропускні здатності для маршруту P_3:

   $P_{3C} = \{ C_{1,8} – C_{8,2} – C_{2,3} – C_{3,6} – C_{6,7} \};$

   $C_{1,8} = 0.3;$

   $C_{8,2} = 0.4;$

   $C_{2,3} = 0.1;$

$C_{3,6} = 0.8$;

$C_{6,7} = 0.3$;

$P_{3C} = \{ 0.3 – 0.4 – 0.1 – 0.8 – 0.3 \}$.

4) Пропускні здатності для маршруту P_4:

$P_{4C} = \{ C_{1,8} – C_{8,2} – C_{2,3} – C_{3,7} \}$;

$C_{1,8} = 0.3$;

$C_{8,2} = 0.4$;

$C_{2,3} = 0.1$;

$C_{3,7} = 0.1$;

$P_{4C} = \{ 0.3 – 0.4 – 0.1 – 0.1 \}$.

Кожен із маршрутів пропускних здатностей містить в своєму шляху пропускну здатність $C_{2,3} = 10\%$ що в свою чергу свідчить про те що всі шляхи є непридатними.

2.4. Порівняння пошуку шляхів від зміни динаміки мережі

Пошук найоптимальнішого шляху будемо як і раніше розглядати на прикладі вершин V_1 та V_7. Основна оцінка буде проводитись незалежно від роботи алгоритму, важливою ознакою порівняння буде зміна маршруту при динамічній зміні завантаження мережі, а саме пропускних здатностей для всього графу. Для початку необхідно знайти найоптимальніший шлях без зміни завантаження мережі, після чого зробити повторний пошук найоптимальнішого шляху вже після динамічної зміни всіх пропускних здатностей на графі.

1) Після пошуку найоптимальнішого маршруту за всіма кроками алгоритму отримаємо наступний маршрут $P_2 = \{ V_1 - V_5 - V_9 - V_2 - V_3 - V_6 \}$. Даний маршрут зображений чорним жирним кольором на графі рис. 2.4.

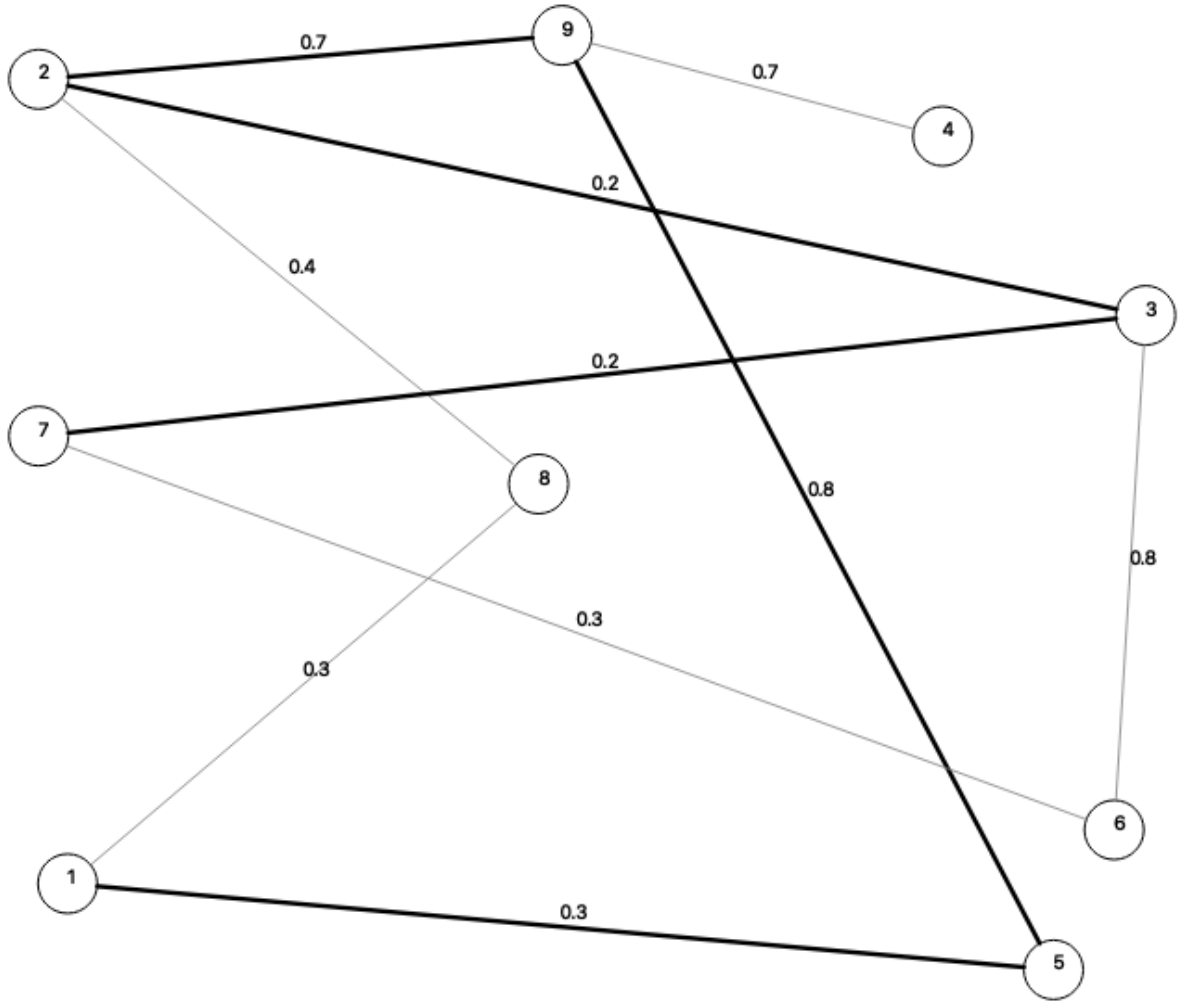


Рис. 2.4. Найоптимальніший маршрут P_2 до динамічної зміни завантаження мережі

Відповідно до цього отримаємо гістограму метрики для знайденого шляху вже із зменшеною пропускною здатністю на кожній ділянці маршруту. Дана гістограма дозволяє відобразити пропускну здатність кожної ділянки маршруту у більш наглядному, а саме у графічному вигляді.

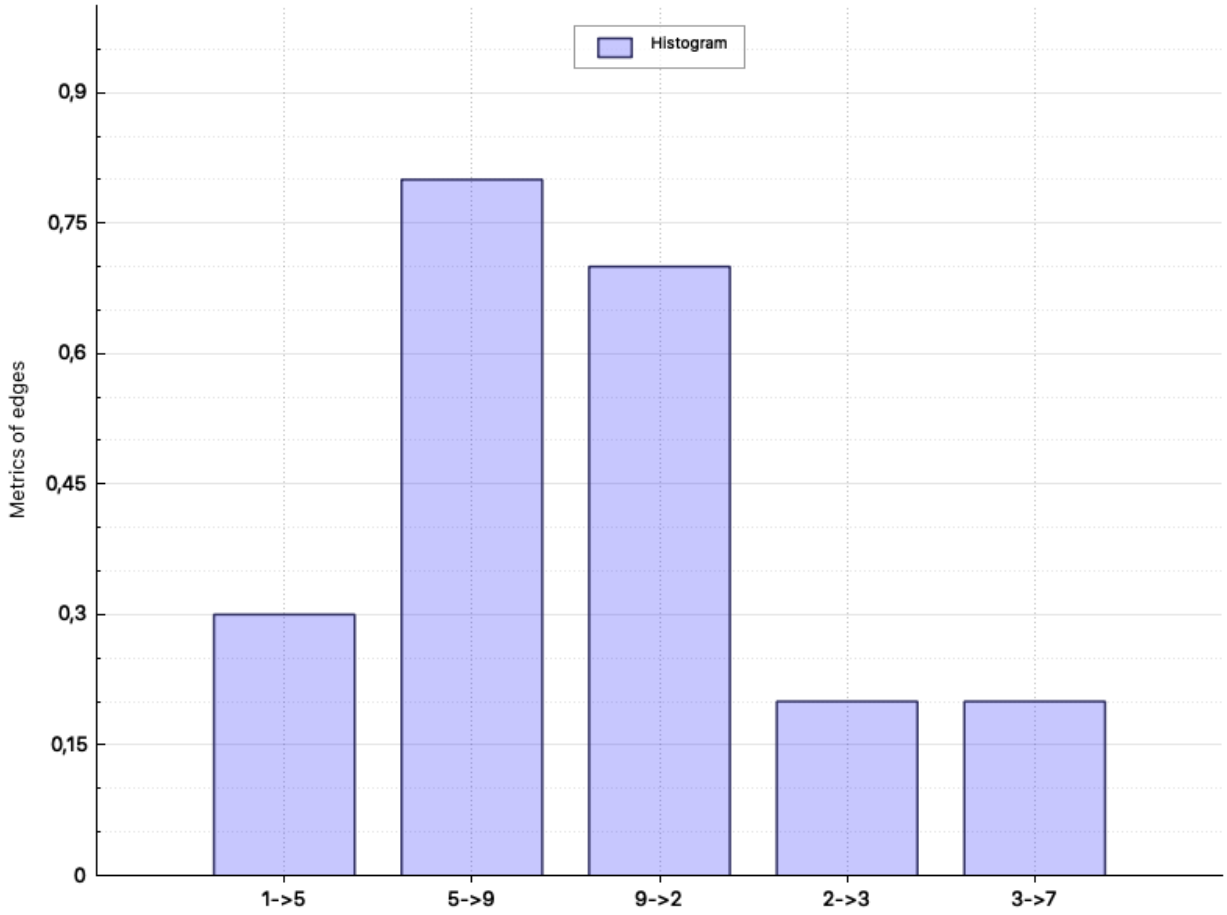


Рис. 2.5. Графічне відображення пропускної здатності кожної з ділянок маршруту до динамічної зміни завантаження мережі

Згідно з гістограми, можемо побачити, що на даному маршруті є дві ділянки з доволі високими значеннями пропускної здатності, а саме ділянки $C_{5,9} = 0.8$ та $C_{9,2} = 0.7$. Сам же маршрут складається з 5 ребер та 6 вершин відповідно.

2) Змінимо пропускну здатність для кожної з ділянок графу та виконаємо пошук найоптимальнішого шляху між вершинами V_1 та V_7. Після пошуку найоптимальнішого маршруту за всіма кроками алгоритму отримаємо наступний маршрут $P_{2D} = \{ V_1 - V_5 - V_9 - V_2 - V_3 - V_6 \}$. Даний маршрут зображений чорним жирним кольором на графі Рисунок 2.6.

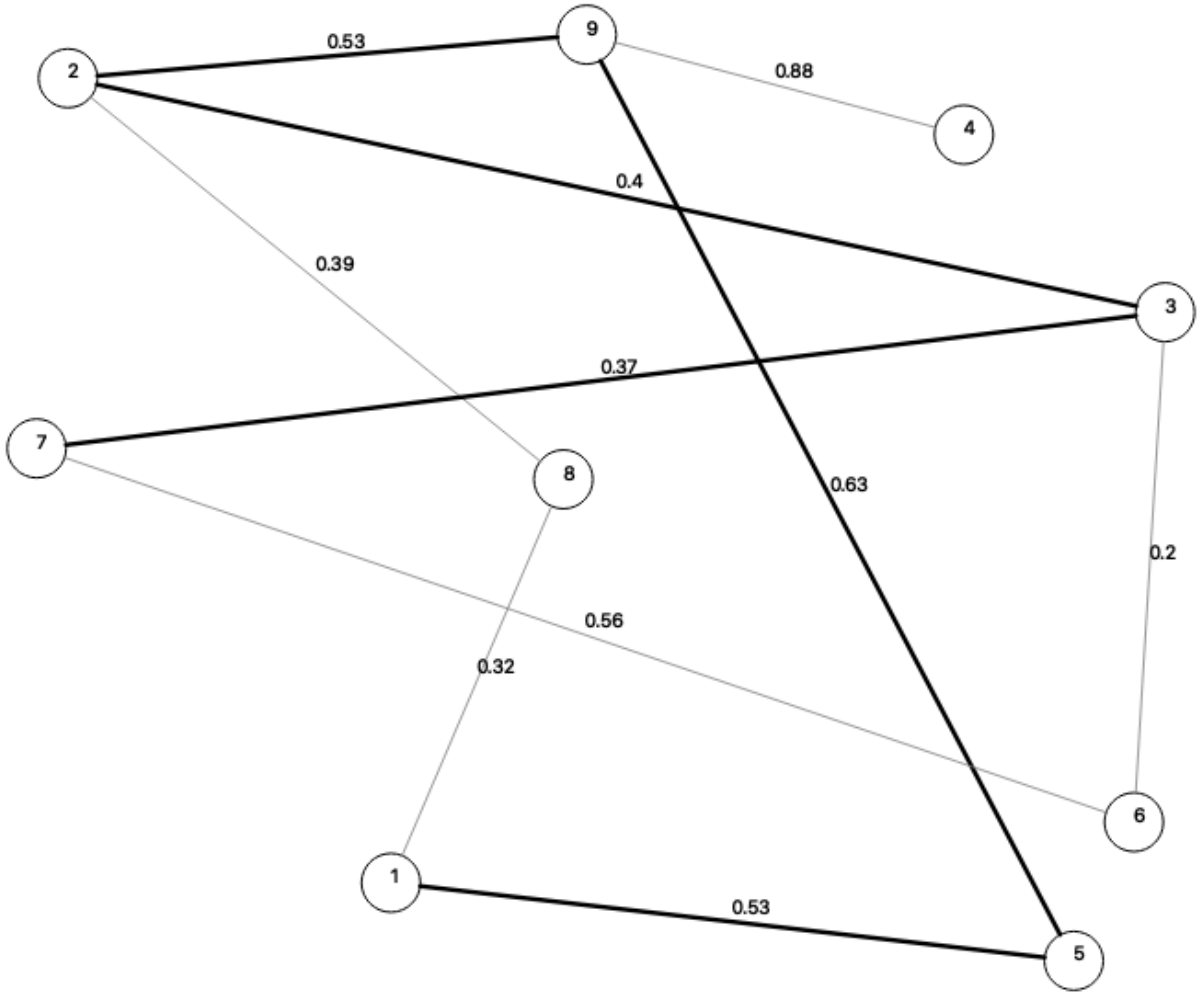


Рис. 2.6. Найоптимальніший маршрут P_{2D} після динамічної зміни завантаження мережі

Згідно з графа зображеного на Рисунку 2.6. видно, що пропускні здатності змінились для всієї мережі проте склад вершин в результуючому маршруті не змінився. Для більшої наглядності розглянемо відповідну гістограму для даного маршруту.

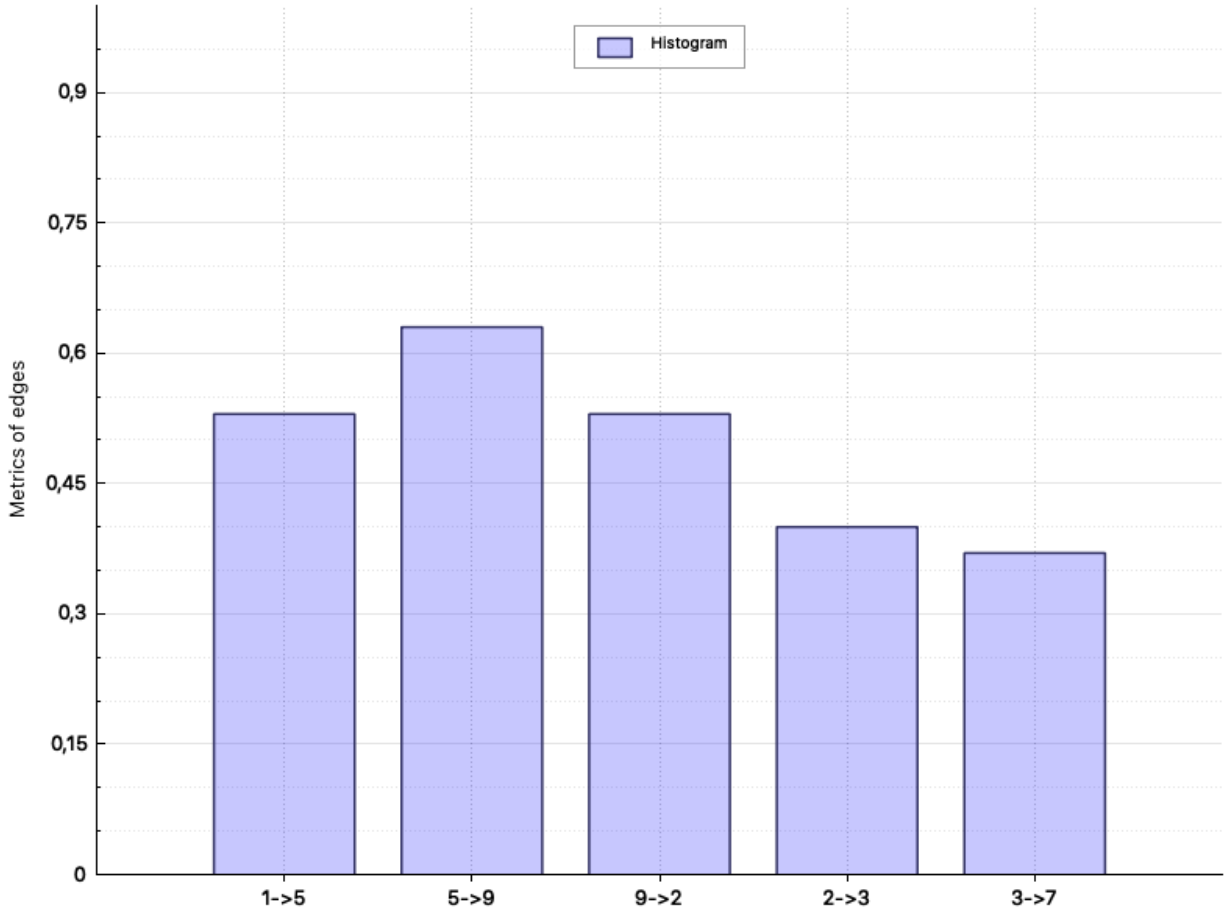


Рис. 2.7. Графічне відображення пропускної здатності кожної з ділянок маршруту після динамічної зміни завантаження мережі

Порівнявши обидві гістограми, можна зробити висновок, що після динамічної зміни завантаження мережі отриманий маршрут складається з більшої кількості ділянок з високою пропускною здатністю в порівнянні зі знайденим маршрутом до динамічної зміни завантаження мережі. Відповідно до отриманих результатів можемо сформувати підсумок та зафіксувати те, що мережа після динамічної зміни всіх пропускних здатностей є більш щільно завантаженою.

ВИСНОВКИ ДО РОЗДІЛУ 2

Вибір найоптимальнішого маршруту залежить від наявності в маршруті найбільшої кількості ділянок з найвищою пропускною здатністю відносно інших маршрутів, а також залежить від кількості ребер в цьому маршруті. Більш оптимальними є шляхи з невеликою кількістю ребер та великою щільністю пропускної здатності на всьому маршруті.

Відповідно до задачі швидкого пошуку найоптимальнішого шляху на графі було запропоновано модифікований алгоритм. Дана модифікація дозволяє розвантажити виконання пошуку найоптимальнішого шляху. Дана особливість досягається за рахунок наявності спеціальної структури, яка з часом накопичує вже знайдені маршрути між певними вершинами. В основі модифікованого алгоритму лежить рекурсивний алгоритм пошуку в ширину. Алгоритм пошуку в ширину використовується лише для пошуку всіх можливих шляхів між двома заданими вершинами. Запропонована модифікація алгоритму дозволяє суттєво прискорити виконання пошуку найоптимальнішого шляху, це підтверджується доволі простою часовою складністю алгоритму пошуку в ширину, а також розвантаженням пошуку в часі за рахунок наявності спеціальної структури для накопичення вже знайдених шляхів.

РОЗДІЛ 3
МОДЕЛЮВАННЯ ЗАПРОПОНОВАНОГО АЛГОРИТМУ

3.1. Огляд технологій

Швидкість виконання алгоритму, як відомо, залежить не тільки від самого алгоритму та його алгоритмічної або ж часової складності, а й від того як він реалізований, а також за допомогою якої мови програмування. Відповідно до потреби швидкої та надійної роботи алгоритму було обрано мову програмування C++ [8] та набір бібліотек Qt [9]. Набір бібліотек Qt (Qt framework) дозволяють писати крос-платформний код, що може бути виконаний на будь-якій операційній системі де встановлений компілятор C++ та набір бібліотек Qt. Дана низькорівнева мова програмування є компільованою, що в свою чергу дозволяє процесору швидко виконувати усі написані інструкції. Окрім вище перерахованого слід розглянути переваги C++ детальніше [10]:

- C++ є відгалуженням мови програмування C, так як був створений на її основі, це в свою чергу дає можливість виконувати написані програми надзвичайно швидко;
- C++ є універсальною мовою програмування так як будь-яка сучасна операційна система містить компілятор C++;
- C++ не є застарілою мовою програмування, він досі підтримується та містить надзвичайно велику кількість бібліотек та фреймворків;
- C++ має велику кількість стандартів, які в свою чергу можна підходять для вирішення будь-якої задачі. З року в рік стандарти оновлюється, за рахунок цього мова програмування C++ поповнюється новими та зручними конструкціями для більш зручнішого та швидшого вирішення різних проблем.

Для розроблення програми було використане програмне середовище Qt Creator та Sublime Text 3. Однак можна використовувати будь-яке інше

середовище для редагування коду написаного на C++. Однією з переваг програмного середовища Qt є компіляція бінарних файлів безпосередньо в даному середовищі, без додаткових утиліт. Для розробки, або ж підтримки програми також можна використовувати текстовий редактор Sublime Text, однак тоді необхідно компілювати бінарні файли через термінал або ж консоль.

Також слід відмітити, що для розробки програми комп'ютерна система повинна мати наступні характеристики:

- Процесор з тактовою частотою не менше 1.25 ГГц;
- оперативна пам'ять не менше 4 Гб;
- встановлений компілятор для C++ з підтримкою стандарту C++11;
- Встановлений набір бібліотек Qt версії 5.13.0 та вище.

Вимоги до інших характеристик не такі вагомі тому можемо ними знехтувати.

3.2. Опис логіки програми

Розроблений програмний продукт повинен надавати необхідні можливості для роботи з перерахованими функціями, що наведені нижче:

- Створення, збереження та редагування графу мережі;
- моделювання запропонованого алгоритму найоптимальнішого шляху мережі;
- перегляд та аналіз результатів виконання розробленої програми.

Відповідно до потреб програми виконувати різні, але залежні одна від одної функції було зауважено, що реалізацію слід розділити на окремі модулі:

- Модуль, що виконує всі маніпуляції над графом та відповідає за його функціонування;
- модуль, що відповідає за серіалізацію та десеарілізацію даних для побудови графу та його збереження;
- модуль, що виконує всі алгоритмічні обчислення;

- модуль, що виконує аналіз алгоритмічних обчислень у вигляді гістограм;
- модуль, що відповідає за інструкції користувача.

Згідно із розглянутих модулів слід розглянути структуру програми. Розглянемо модуль з назвою CDN, що відповідає за всі алгоритмічні обчислення, до його складу входять класи CDN та Matrix.

Клас Matrix відповідає за представлення графа у вигляді матриці та містить набір відповідних функцій та методів:

- метод transformFrom – дозволяє перетворити граф з графічної форми в математичну, а саме в матрицю переходів, де перетин вершин показує пропускну здатність між цими вершинами;
- метод setRowsCount – дозволяє встановити значення кількості рядків матриці;
- метод setColsCount – дозволяє встановити значення кількості стовпців матриці;
- функція getRowsCount – дозволяє отримати значення кількості рядків матриці;
- функція getColsCount – дозволяє отримати значення кількості стовпців матриці.

Клас CDN відповідає за всі алгоритмічні обчислення, в основі яких лежить робота над класом Matrix, обчислення виконуються за допомогою наступних функцій та методів:

- функція getMatrix – дозволяє отримати сформовану матрицю переходів у вигляді списку списків;
- функція getLastFoundPath – дозволяє отримати останній знайдений найоптимальніший шлях на графі у вигляді списку вершин;
- функція getMetricsOfLastFoundPath – дозволяє отримати пропускні здатності на останньому знайденому

найоптимальнішому шляхові у вигляді списку пропускних здатностей;

- метод setMatrix – дозволяє встановити сформовану матрицю для подальших обчислень;
- метод addPath – дозволяє зберегти один із знайдених шляхів на графі;
- функція findPath – дозволяє знайти та отримати найоптимальніший шлях у вигляді списку вершин;
- метод findPathsUtil – дозволяє рекурсивно знайти всі можливі шляхи на графі між двома заданими вершинами;
- метод calcMetricsOfLastFoundPath – дозволяє порахувати метрику для останнього знайденого найоптимальнішого шляху на графі;
- метод resetSettings – дозволяє скинути всі налаштування для даного класу та очистити всі додаткові поля класу.

Наступним слід розглянути модуль з назвою Parse, що відповідає за серіалізацію та десеарілізацію даних, іншими словами модуль містить класи, що в свою чергу дозволяють зберігати конфігурацію графу мережі у вигляді json файлу, а також зчитувати вже створену конфігурацію з json файлу. Можливість модифікування графу є доступною безпосередньо через json файл. До даного модулю входять класи JsonReader та JsonWriter.

Клас JsonReader відповідає за зчитування створеної конфігурації графа із відповідного файлу, зчитування інформації із файлу відбувається за допомогою наступних методів:

- метод readFromJson – дозволяє зчитати конфігурацію графу із json файлу;
- метод setName – дозволяє встановити ім’я файлу для зчитування конфігурації.

Клас JsonWriter відповідає за збереження поточної конфігурації графа, збереження інформації відбувається за допомогою запису інформації у файл із розширенням json. Запис даних відбувається за допомогою наступних методів:

- метод writeToJson – дозволяє записати поточну конфігурацію графу у json файл;
- метод setName – дозволяє встановити ім’я файлу;
- метод setNodes – дозволяє встановити список вершин графу для зчитування з них поточної конфігурації та подальшого запису у json файл.

Наступним слід розглянути модуль із назвою Graph, що в свою чергу відповідає за візуалізацію усіх операції над графом. Даний модуль складається з класів Node, Edge та GraphWidget.

Клас Node представляє собою вершину графу та складається з наступних, притаманних вершині графа функцій та методів:

- метод addEdge – дозволяє додати вершину графа;
- функція edges – дозволяє повернути список усіх можливих ребер для даної вершини;
- метод calculateForces – дозволяє розрахувати всі необхідні логічні характеристики елемента вершини для розташування її на сцені;
- функція advancePosition – дозволяє оновити усі необхідні дані щодо позиціонування даної вершини;
- функція boundingRect – дозволяє створити необхідний каркас для відображення даної вершини у вигляді прямокутника;
- функція shape – дозволяє створити візуальне представлення вершини у вигляді еліпсу;
- метод paint – дозволяє намалювати відповідне та точне графічне графічне представлення вершини;
- функція itemChange – дозволяє змінити позицію усіх сполучених з нею ребер;

- метод mousePressEvent – дозволяє обробити усі події натискання миші;
- метод mouseReleaseEvent – дозволяє обробити усі події після того, як кнопка миші була відпущена.

Клас Edge представляє собою ребро графа та складається з наступних, притаманних ребру графа функцій:

- функція sourceNode – дозволяє отримати початкову вершину ребра;
- функція destNode – дозволяє отримати кінцеву вершину ребра;
- метод adjust – дозволяє скорегувати координати ребра;
- функція getMetric – дозволяє повернути пропускну здатність;
- метод setColor – дозволяє встановити колір зафарбування ребра;
- метод setPenWidth – дозволяє встановити товщину ребра;
- метод setMetric – дозволяє встановити пропускну здатність для даного ребра;
- функція boundingRect – дозволяє створити необхідний каркас для відображення даного ребра;
- метод paint – дозволяє намалювати відповідне та точне графічне представлення вершини;

Клас GraphWidget представляє собою графічну сцену для відображення графа та усіх можливих операцій над ним. Слід розглянути основні функції даного класу:

- метод shuffle – дозволяє випадковим чином розмістити усі компоненти графу на графічній сцені;
- метод zoomIn – дозволяє збільшити розмір графа на сцені;
- метод zoomOut – дозволяє зменшити розмір графа на сцені;
- метод keyPressEvent – є однією із найголовніших функцій даного класу, що дозволяє відповідно до натиснутої кнопки виконати конкретну команду;
- метод wheelEvent – дозволяє постійно оновлювати графічну сцену;

- метод scaleView – дозволяє оновлювати усі необхідні дані для графічної сцени.

Останнім модулем із розглянутих є модуль із назвою Info. Даний модуль відповідає за відображення інструкцій користувача та аналіз оброблених алгоритмом даних у вигляді гістограм. Відповідний модуль складається із класів Instructions та HistogramConstructor.

Клас Instructions є допоміжним класом і дозволяє відобразити інструкції користувача у окремому вікні. Показ усіх можливих інструкцій виконується за допомогою функції showInstructions. Для показу всіх можливих інструкцій необхідно набрати на клавіатурі слово HELP англійськими літерами.

Клас HistogramConstructor є також допоміжним класом, що дозволяє проводити аналіз порівняння знайденого найоптимальнішого шляху зі знайденим найоптимальнішим шляхом після динамічної зміни графу мережі. Даний клас надає можливість виконання операцій відображення, збереження та видалення створеної гістограми. Усі перераховані операції виконуються за допомогою наступних функцій:

- метод showHistogram – дозволяє відобразити створену гістограму;
- метод saveHistogram – дозволяє зберегти створену гістограму;
- метод clearHistogram – дозволяє очистити усі конфігурації для знайденої гістограми.

ВИСНОВКИ ДО РОЗДІЛУ 3

У даному розділі було розглянуто моделювання запропонованого алгоритму. Було наведено ряд сучасних технологій та мов програмування, що були використані для розробки алгоритму. Також було наведено низку характеристик, що вимагаються для моделювання розглянутого алгоритму.

У другому підрозділі було розглянуто набір модулів програмної реалізації, що дозволяють моделювати запропонований алгоритм. Відповідно до наведених модулів було детально розглянуто набір усіх методів та функцій кожного з модулів, що поєднані у класи, та дозволяють описати не тільки логіку алгоритму, а й реалізувати додатковий функціонал для демонстрації роботи самого алгоритму. Відповідно до проаналізованого слід зазначити, що кожен з модулів відповідає за конкретний функціонал, що дозволило розглянути алгоритм, як набір незалежних модулів, які при поєднанні виконують необхідне моделювання алгоритму.

РОЗДІЛ 4
ОГЛЯД РОЗРОБЛЕНОГО ЗАСТОСУНКУ

4.1. Опис програмного інтерфейсу

Відповідно до задачі пошуку найоптимальнішого маршруту на графі мережі було вирішено розробити гнучкий інтерфейс з необхідним функціоналом. Даний інтерфейс зобов'язаний бути зрозумілим для користувача та бути зручним у використанні. Згідно з перерахованих вимог було розроблено відповідний програмний інтерфейс. Готова реалізація програмного продукту забезпечена усіма необхідними функціями для демонстрації роботи запропонованого алгоритму. Як приклад готового реалізованого програмного інтерфейсу необхідно розглянути увесь функціонал на прикладі. Основне вікно готового продукту продемонстровано на рисунку 3.1.

Рисунок 3.1. Основне вікно програми

Слід відмітити, що дане вікно не має жодного меню та інших допоміжних кнопок, оскільки весь функціонал доступний після набору на клавіатурі відповідної команди. Розглянемо кожну з команд на окремому прикладі. Програмна реалізація дозволяє обробити будь-які команди в будь-якій послідовності, однак як приклад буде продемонстровано правильне використання команд в правильній послідовності.

Першою слід розглянути команду “OPEN”. Дана команда дозволяє відкрити вікно вибору файлу конфігурації графа. Після вибору файлу, програма автоматично завантажить конфігурацію графа та відобразить його на графічній сцені. Можемо розглянути приклад вікна вибору файлу на рисунку 3.2.

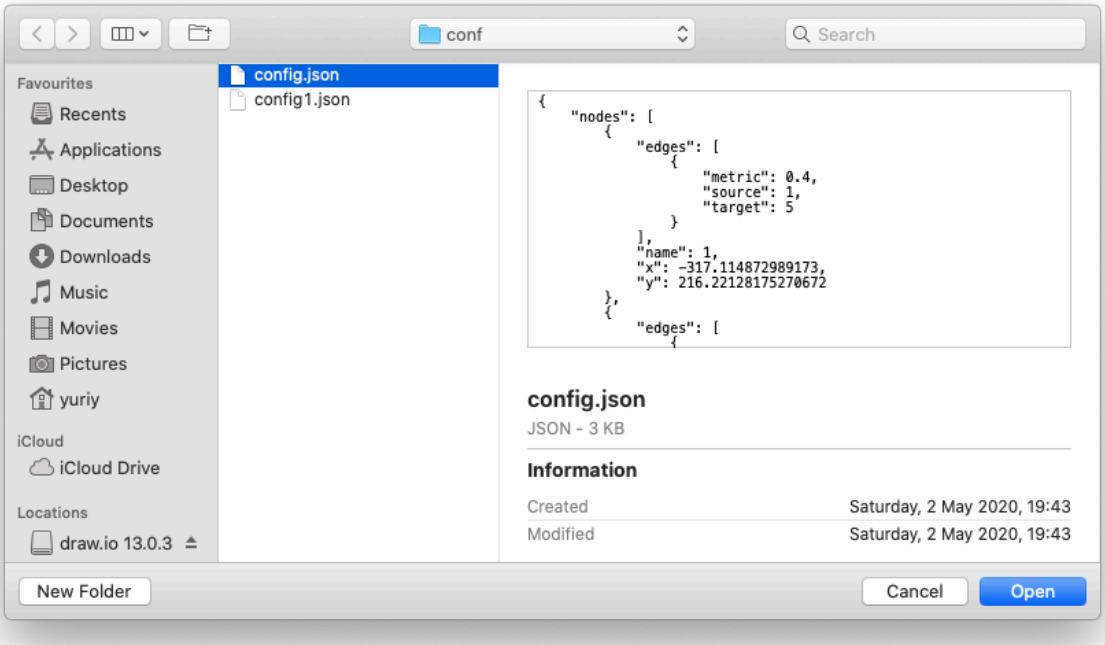


Рисунок 3.2. Вікно вибору файлу конфігурації графу

Після успішного завершення вибору файлу конфігурації графу мережі, програма дозволяє відобразити дану конфігурацію на графічній сцені. Приклад завантаженого графа на графічну сцену відображено на рисунку 3.3.

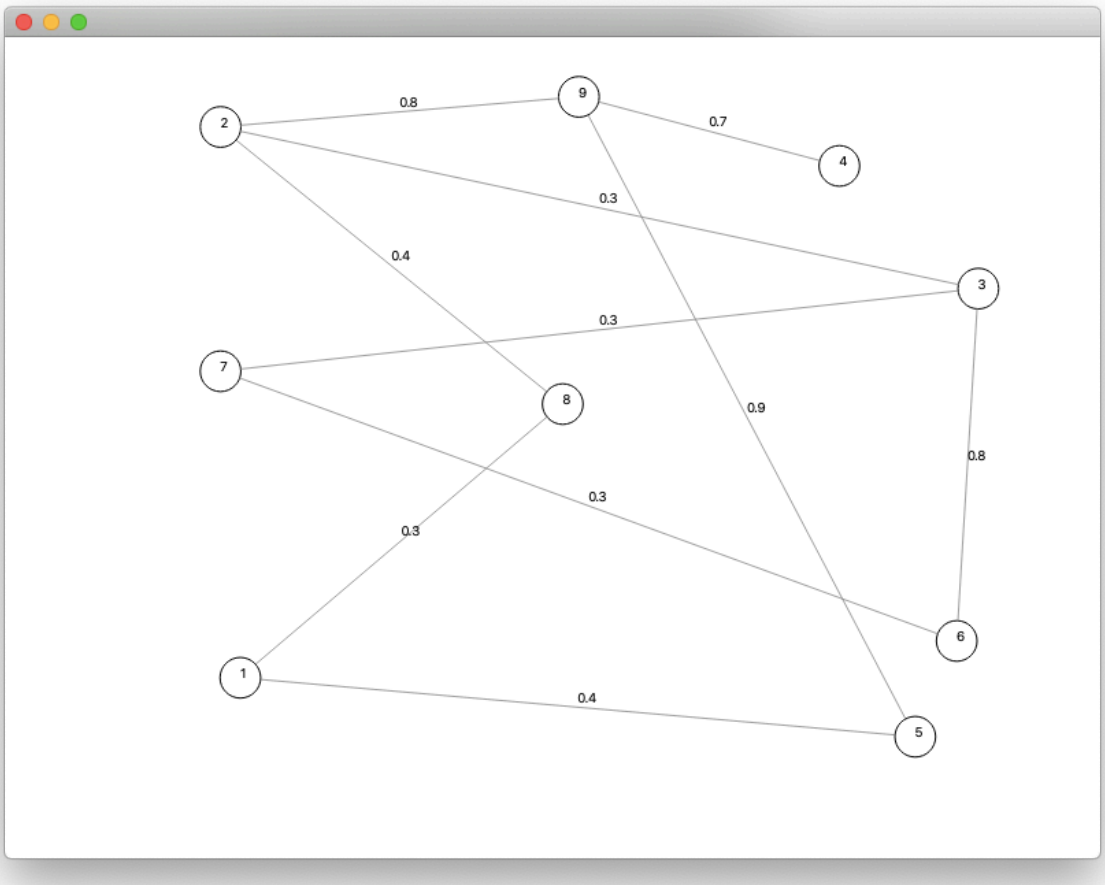


Рисунок 3.3. Вікно відображення графу на графічній сцені

Наступною слід розглянути команду “CALC”. Дана команда дозволяє виконати пошук найоптимальнішого шляху на графі та відобразити його на графічній сцені. Після набору даної команди на клавіатурі необхідно ввести початкову та кінцеву вершину для пошуку у спеціальне вікно. Приклад вікна введення вершин можемо розглянути на рисунку 3.4. Після введення коректних вершин графа, програма виконає пошук найоптимальнішого шляху, відповідно до розглянутого у розділі 2 алгоритму та відобразить результат на графічній сцені. Приклад знайденого найоптимальнішого маршруту на графі мережі розглянуто на рисунку 3.5.

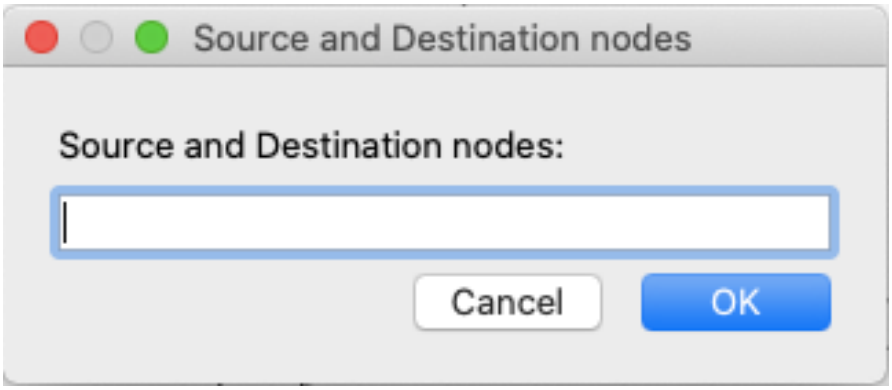


Рисунок 3.4. Вікно введення вершин графу

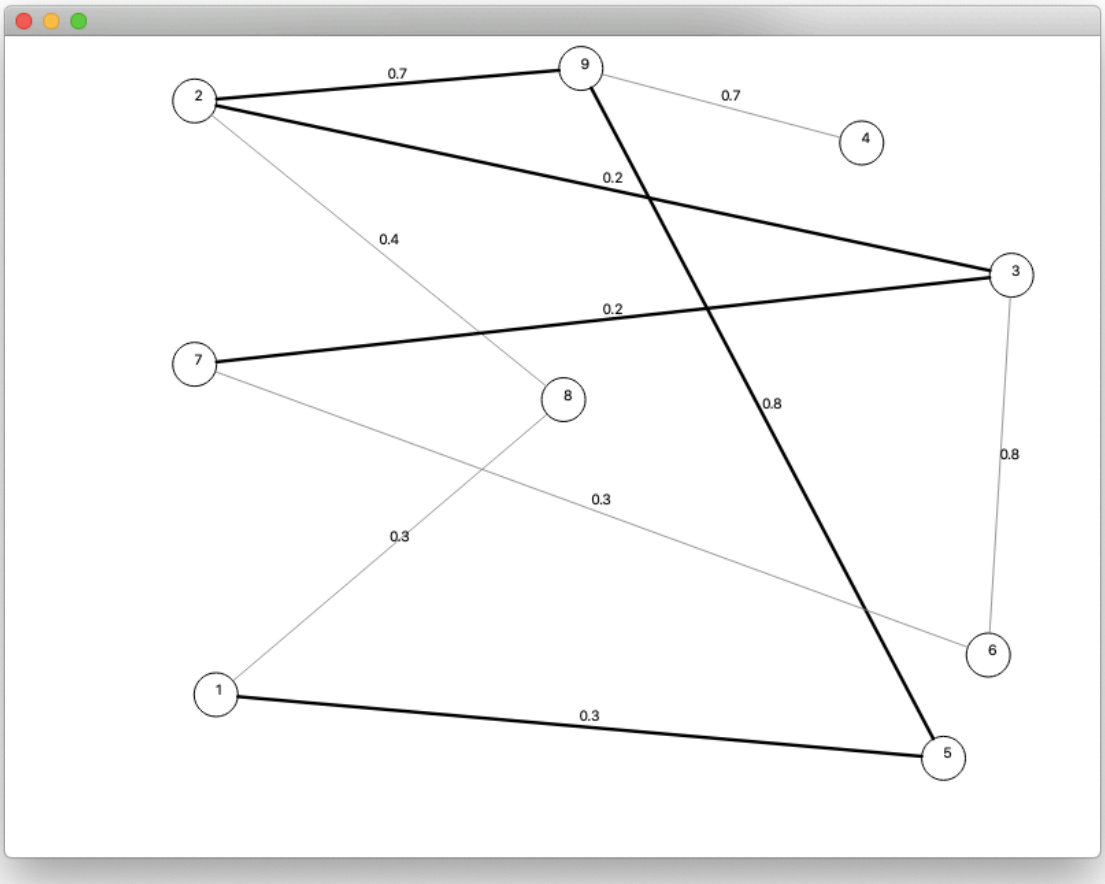


Рисунок 3.5. Вікно відображення найоптимальнішого шляху

Наступною командою для розгляду є команда “HISTOGRAM”. Дана команда дозволяє створити гістограму відповідно до знайденого найоптимальнішого шляху та відобразити її у спеціальному вікні. Створена

гістограма дозволяє у зручному вигляді відобразити пропускні здатності знайденого шляху та оброблені вершини графу. Приклад вікна відображення гістограми можемо розглянути на рисунку 3.6.

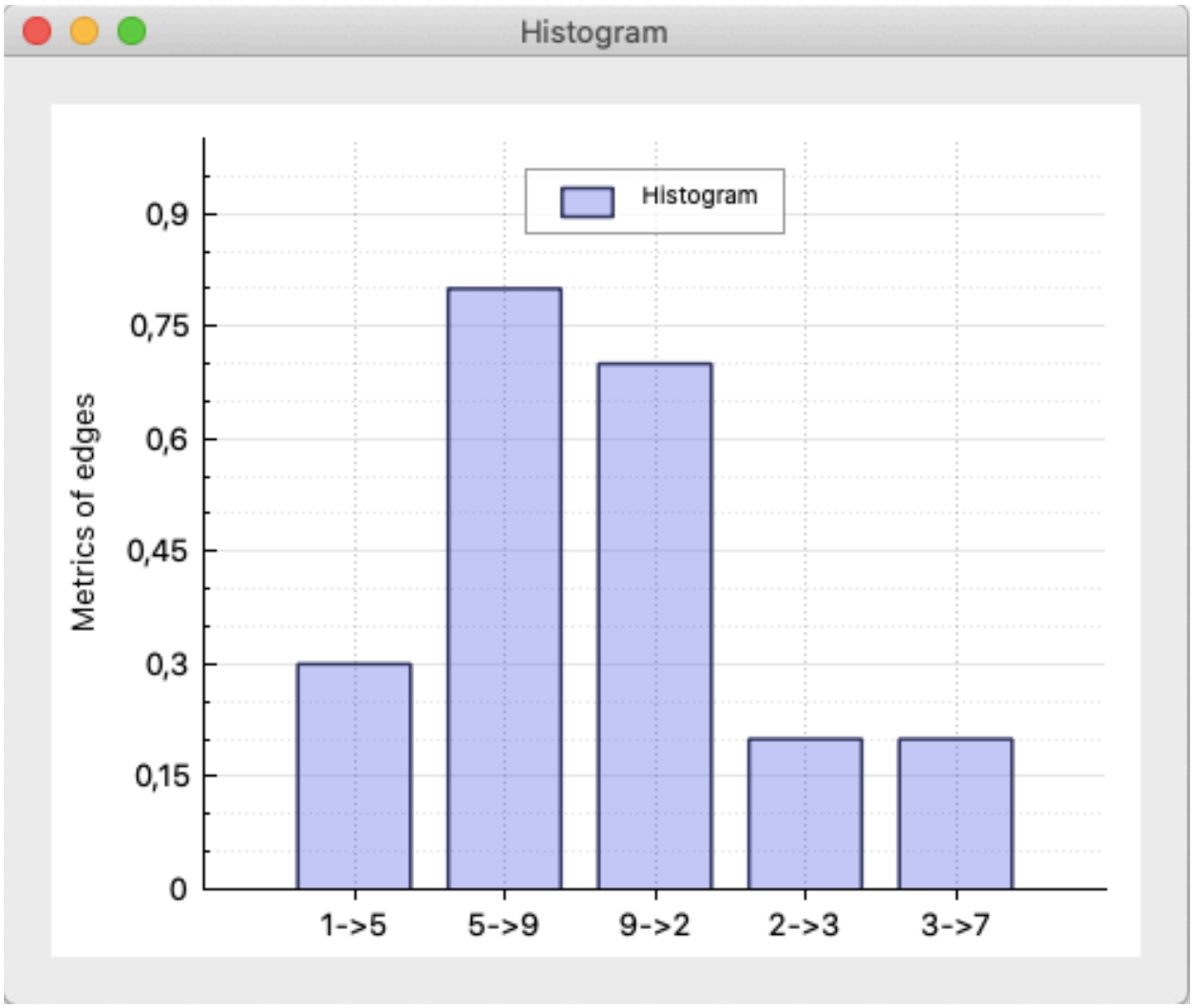


Рисунок 3.7. Вікно відображення створеної гістограми

Однією із додаткових команд є команда “RAND”, що в свою чергу дозволяє випадковим заповненням динамічно змінити пропускні здатності для всього графу мережі. Приклад результату роботи команди можемо розглянути на рисунку 3.8.

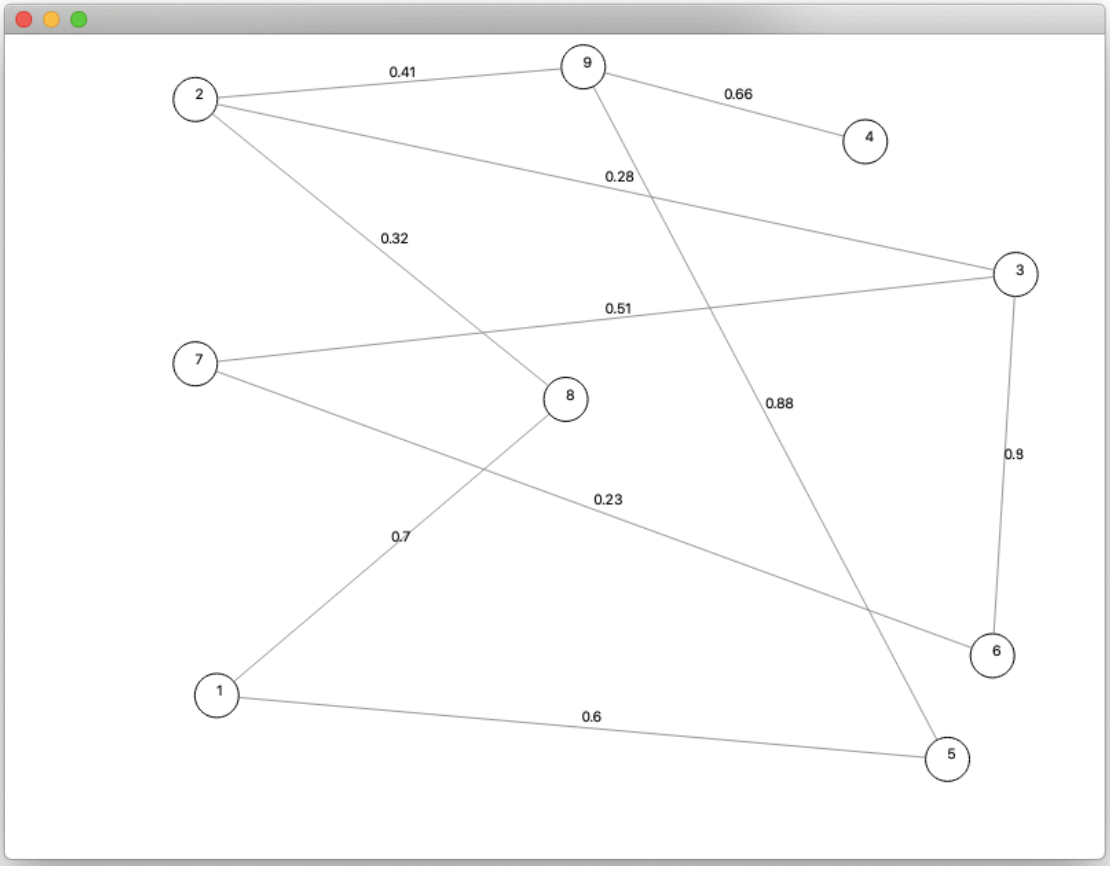


Рисунок 3.8. Вікно основної графічної сцени після динамічної зміни пропускних здатностей

Слід також розглянути додаткову команду, що дозволяє переглянути увесь список доступний команд. Дана команда доступна після набору на клавіатурі комбінації букв “HELP”. Приклад вікна усіх доступних команд можна розглянути на рисунку 3.9.

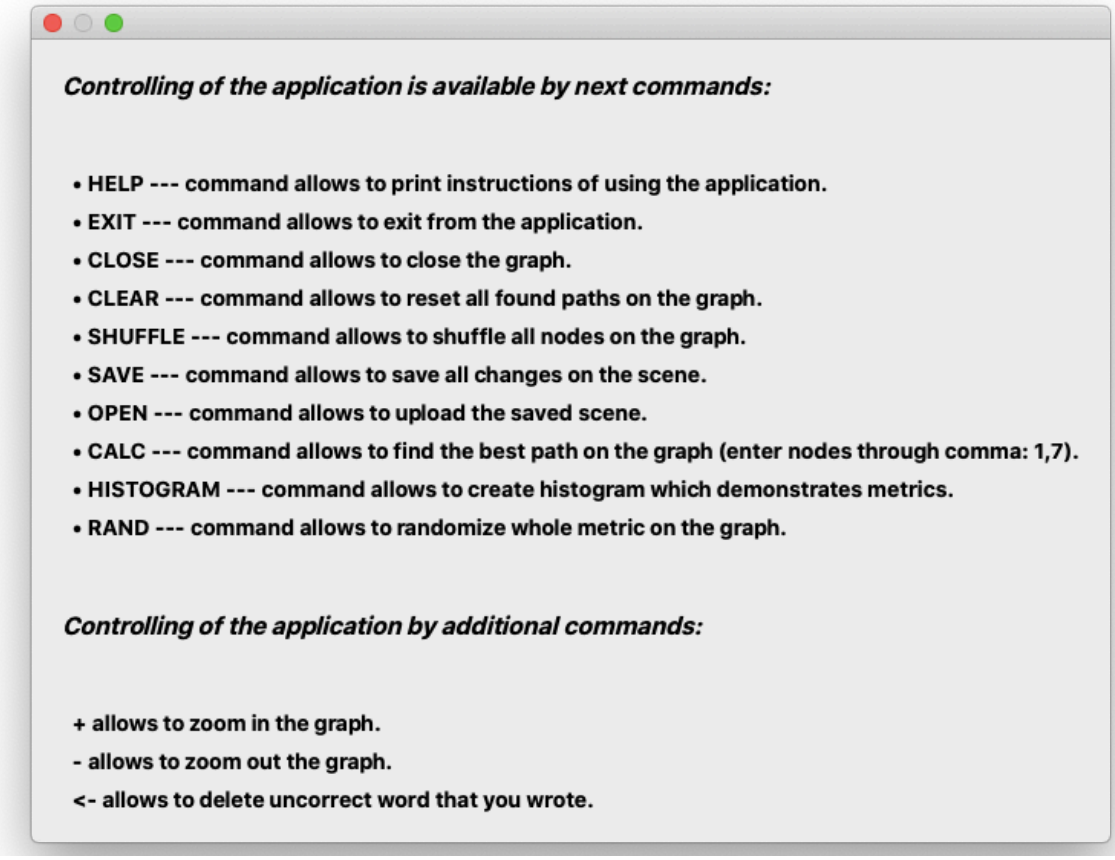


Рисунок 3.9. Вікно усіх можливих команд

Останньою з основних команд є команда “SAVE”. Дана команда дозволяє зберегти поточну конфігурацію графу мережі у конкретний файл. Графічне вікно вибору файлу для збереження поточної конфігурації графу мережі є таким же як і вікно вибору файлу для зчитування конфігурації графу мережі. Відповідний приклад був розглянутий на рисунку 3.2.

Необхідно також звернути увагу на додаткові команди, які слугують допоміжними та дають можливість полегшити роботу користувача з графом. Нижче наведений перелік таких команд з описом їх функціоналу:

- команда “SHUFFLE” – дозволяє змінити положення вершин графу на графічній сцені випадковим чином. Приклад роботи даної команди продемонстрований на рисунку 3.10;
- команда “CLEAR” – дозволяє очистити з графічної сцени

останній знайдений найоптимальніший шлях. Приклад виконання даної команди можна розглянути на рисунку 3.11;

- команда “CLOSE” – дозволяє видалити граф з графічної сцени та видалити усі відповідні налаштування. Після виконання даної команди вікно програми буде відображати лише пусту графічну сцену. Приклад такого вікна було розглянуто на рисунку 3.1.

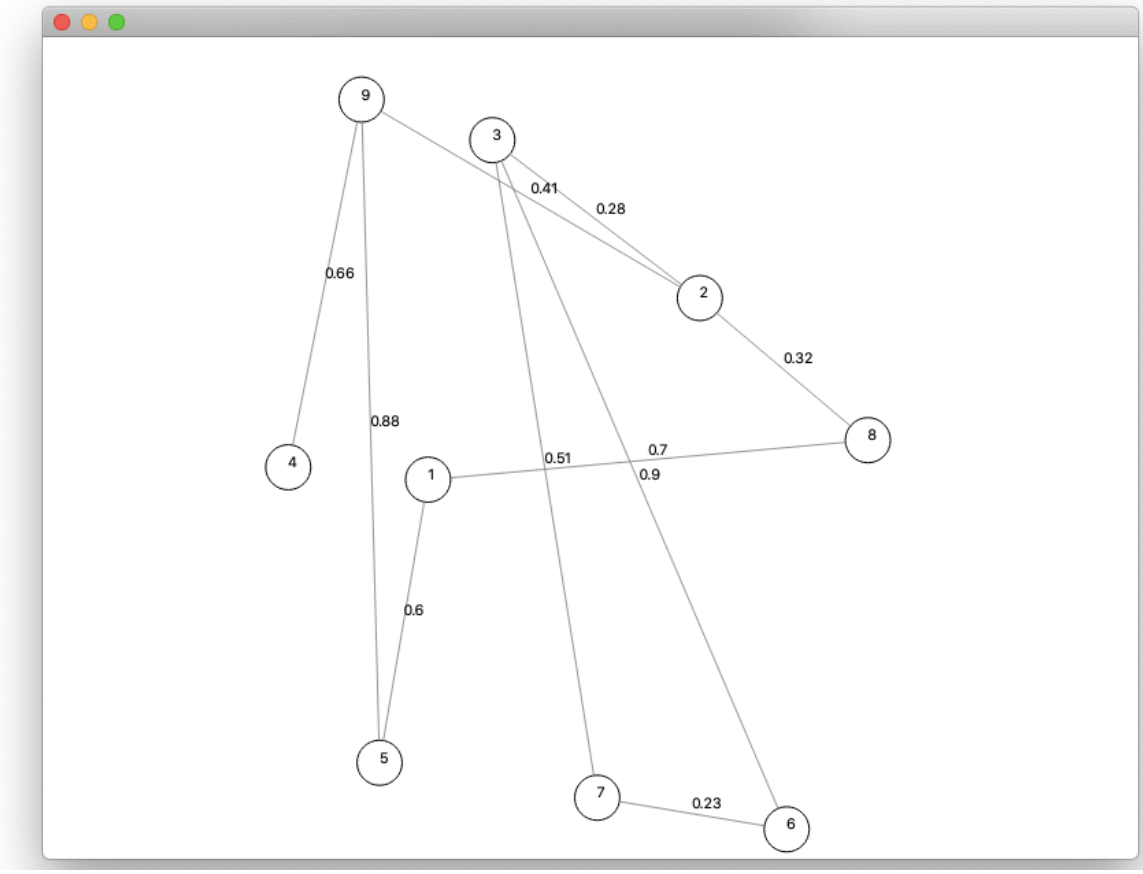


Рисунок 3.10. Результат виконання команди “SHUFFLE”

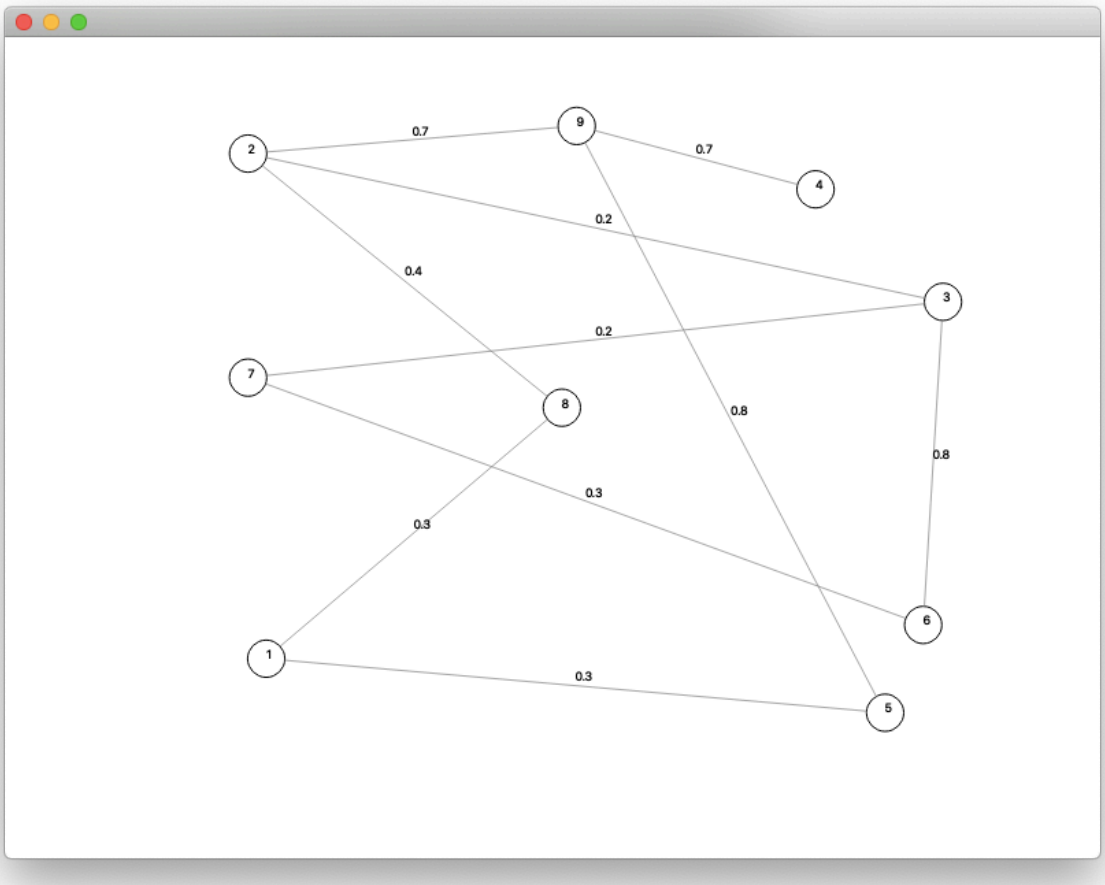


Рисунок 3.11. Результат виконання команди “CLEAR”

Окрім усіх вище розглянутих команд, також слід звернути увагу на ряд допоміжних команд, які виконують лише конкретні додаткові функції, які ніяким чином не стосуються виконання алгоритму або ж операцій пов’язаних з ним. До таких команд можна віднести команди та операції:

- команда “EXIT” – дозволяє закрити вікно програми. Після набору даної команди на клавіатурі, програма автоматично завершить своє виконання у нормальному режимі;
- додаткова операція збільшення розміру графу на графічній сцені. Дана операція доступна за допомогою прокручування колеса миші уверх та натиснутої клавіші Control;
- додаткова операція зменшення розміру графу на графічній сцені.

Дана операція доступна за допомогою прокручування колеса миші вниз та натиснутої клавіші Control;

- додаткова операція видалення неправильно введеної комбінації.

Дана операція доступна за рахунок натискання клавіші BackSpace.

4.2. Розгляд прикладу роботи програми

Слід розглянути приклад роботи готового застосунку на прикладі графу, який продемонстрований на рисунку 3.12. Для початку необхідно створити новий файл конфігурації графу мережі. Для цього слід скористатись будь-яким текстовим редактором та створити нову конфігурацію слідуючи структурі тестового файлу конфігурації, що постачається разом з програмою. Слід відмітити, що даний файл повинен бути створений та поширюватись лише з розширенням json. Після успішного створення файлу конфігурації необхідно завантажити даний файл. Для цього слід набрати на клавіатурі команду “OPEN”, після чого необхідно вибрати попередньо створений файл. Після успішного завантаження файлу конфігурації графу мережі, як результат на графічній сцені має з’явитись сконфігурований граф мережі. Після усіх успішно виконаних кроків необхідно розглянути процедуру пошуку найоптимальнішого маршруту на графі. Для цього необхідно набрати на клавіатурі команду “CALC”, розглянуту у розділі 3.2. Після цього програма запропонує ввести дві вершини для пошуку, початкову та кінцеву. Приклад вікна вводу вершин розглянуто на рисунку 3.13. Для демонстрації роботи пошуку найоптимальнішого шляху було використано вершини 1 та 7.Після вводу запропонованих вершин, програма запустить алгоритм пошуку найоптимальнішого маршруту та відобразить знайдений результат. Результат знайденого найоптимальнішого маршруту продемонстрований на рисунку 3.14.

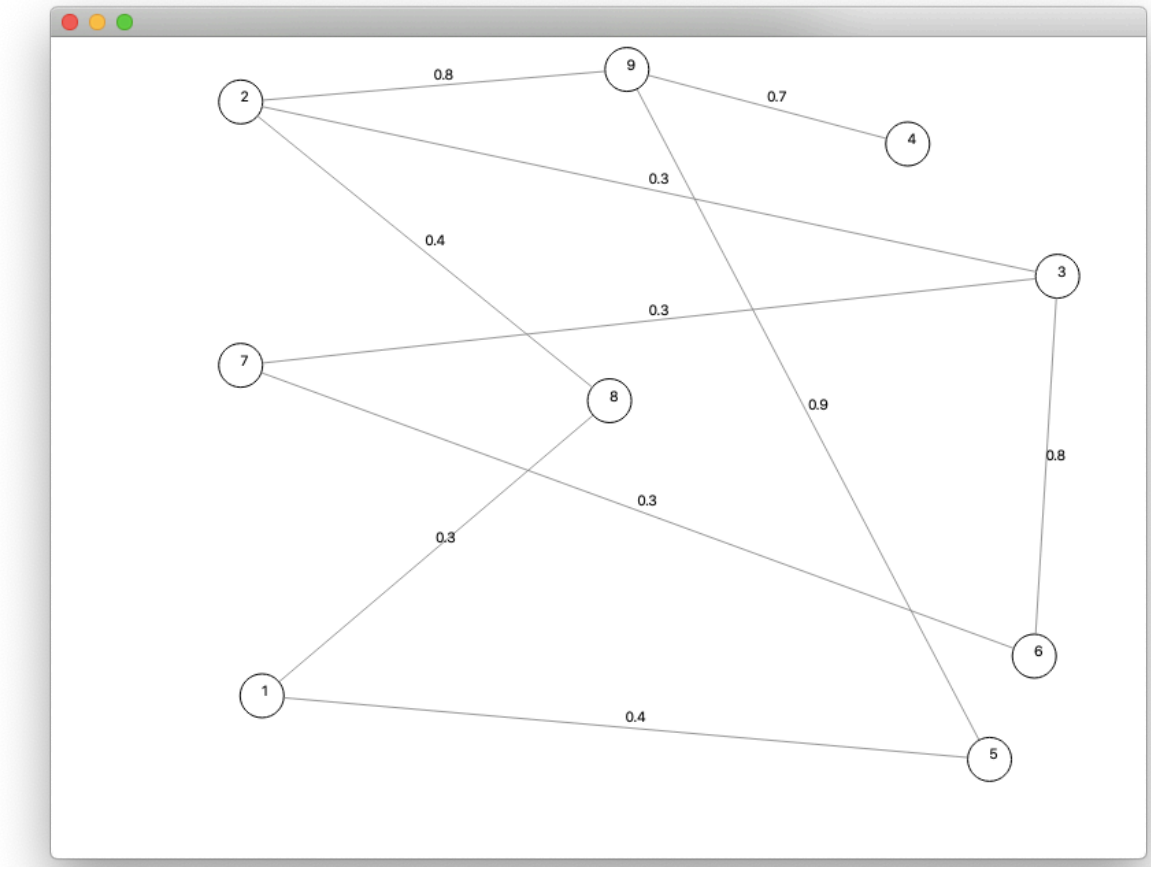


Рисунок 3.12. Граф мережі

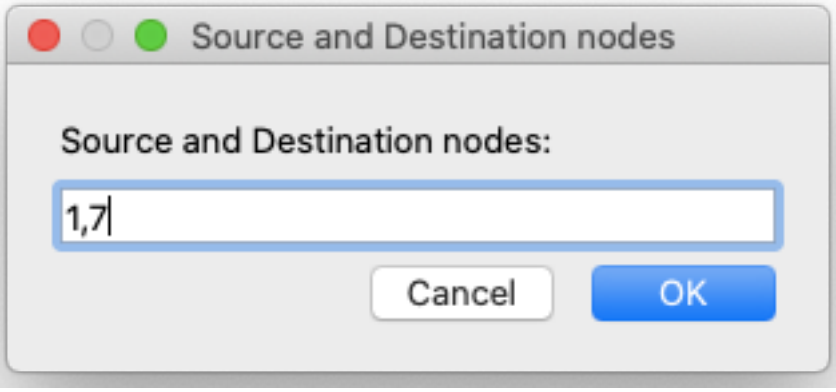


Рисунок 3.13. Вікно вводу вершин

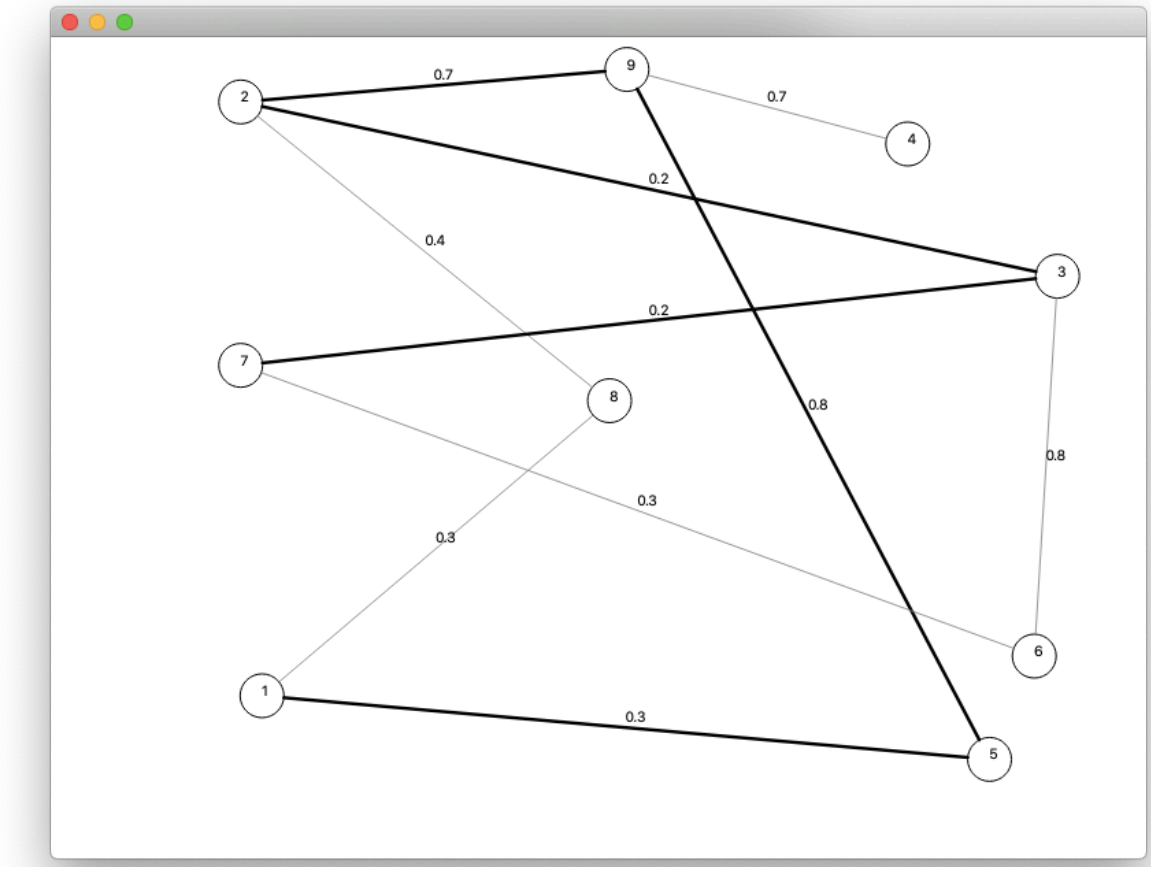


Рисунок 3.14. Вікно відображення найоптимальнішого шляху

Слід також розглянути демонстрацію знайденого шляху у більш наглядному вигляді. Для цього слід скористатись вікном, що демонструє знайдений маршрут у вигляді гістограми. Для відображення вікна гістограми необхідно набрати на клавіатурі команду “HISTOGRAM”. Дана гістограма відображає найоптимальніший маршрут у вигляді стовпців, що відображають переходи між вершинами для знайденого маршруту. Додатковою можливістю гістограми є відображення пропускної здатності кожної з ділянок маршруту. Відповідні пропускні здатності для кожної з ділянок шляху можна розглянути на лівій шкалі гістограми. Результат вікна знайденої гістограми відповідно до попередньо знайденого найоптимальнішого маршруту між вершинами 1 та 7, можна розглянути на рисунку 3.15.

Рисунок 3.15. Вікно відображення гістограми

ВИСНОВКИ ДО РОЗДІЛУ 4

У цьому розділі було продемонстровано розроблений програмний застосунок та розглянуто відповідний гнучкий та зручний інтерфейс даного застосунку. Також було наведено детальний приклад роботи розробленого додатку з відповідними інструкціями щодо користування.

У першому підрозділі було наведено усі можливі інструкції користувача, що були реалізовані за допомогою введення спеціальних команд з клавіатури. Кожна з продемонстрованих інструкцій відповідає за окрему команду та окреме вікно відображення, що місить свій функціонал.

У другому підрозділі було наведено приклад роботи створеного застосунку. Приклад роботи був продемонстрований у конкретній послідовності – притаманній нормальному функціонуванню даного додатку. Прикладом відображення роботи програми слугує демонстрація конкретних зображень, що відображають відповідний результат виконання кожної з команд з кінцевим результатом.

ВИСНОВКИ

Основною метою щодо написання дипломного проєкту була запропонована ідея реалізації власного алгоритму маршрутизації для транспортних мереж. Поставленою ціллю даного алгоритму була вимога ефективно та за константний час вирішувати задачу маршрутизації в транспортних мережах.

В першому розділі було наведено огляд транспортних мереж і методів їх моделювання. Також в деталях було розглянуто усі методи маршрутизації, притаманних даним мережам. Відповідно до розглянутих методів маршрутизації було проаналізовано та порівняно усі найбільш відомі та ефективні алгоритми маршрутизації, що дозволяють вирішити поставлену задачу маршрутизації в транспортних мережах.

Реалізація власного алгоритму маршрутизації була наведена у другому розділі. Було детально продемонстровано роботу створеного алгоритму з чітким математичним описом та прикладами щодо його функціонування.

В третьому розділі було виконано опис усіх використаних технологій та мов програмування, що були використані при розробці наведеного алгоритму маршрутизації та програмного інтерфейсу, який в свою чергу був розроблений для забезпечення організації моделювання даного алгоритму. Також було продемонстровано огляд усіх унікальних та незалежних модулів, їх класів та методів і функцій кожного з класів.

У останньому, четвертому розділі було детально розглянуто інструкції користувача та виконано демонстрацію роботи алгоритму та наведено усі необхідні приклади виконання розробленого застосунку.

Розроблений алгоритм маршрутизації дасть змогу швидко та ефективно вирішувати задачу маршрутизації в транспортних мережах. Використання даного алгоритму є доступним за допомогою реалізованого програмного інтерфейсу, що дозволяє ефективно виконувати моделювання алгоритму.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Транспортна мережа - Wikipedia [Електронний ресурс]. Режим доступу: https://uk.wikipedia.org/wiki/Транспортна_мережа (дата звернення 14.04.2020)
2. Моделювання мережі маршрутів – Eprints.Kname [Електронний ресурс]. Режим доступу: http://eprints.kname.edu.ua/46578/1/ilovepdf_com-141-144.pdf (дата звернення 15.04.2020)
3. Маршрутизація – Znanius [Електронний ресурс]. Режим доступу: http://www.znanius.com/3820.html (дата звернення 17.04.2020)
4. Алгоритмы маршрутизации – Studme [Електронний ресурс]. Режим доступу: https://studme.org/94337/informatika/algoritmy_marshrutizatsii (дата звернення 19.04.2020)
5. Поиск в ширину – Wikipedia [Електронний ресурс]. Режим доступу: https://ru.wikipedia.org/wiki/Поиск_в_ширину (дата звернення 20.04.2020)
6. Print all paths from a given source to a destination – Geeksforgeeks [Електронний ресурс]. Режим доступу: https://www.geeksforgeeks.org/find-paths-given-source-destination/ (дата звернення 22.04.2020)
7. Теорія графів – Wikipedia [Електронний ресурс]. Режим доступу: https://uk.wikipedia.org/wiki/Теорія_графів (дата звернення 23.04.2020)
8. C++ - Wikipedia [Електронний ресурс]. Режим доступу: https://uk.wikipedia.org/wiki/C++ (дата звернення 24.04.2020)
9. Qt – Qt.Io [Електронний ресурс]. Режим доступу: https://www.qt.io/ (дата звернення 27.04.2020)
10. Огляд і основи мови програмування C++ - Znannya [Електронний ресурс]. Режим http://www.znannya.org/?view=Cpp_basics (дата звернення 28.04.2020

ДОДАТОК 1

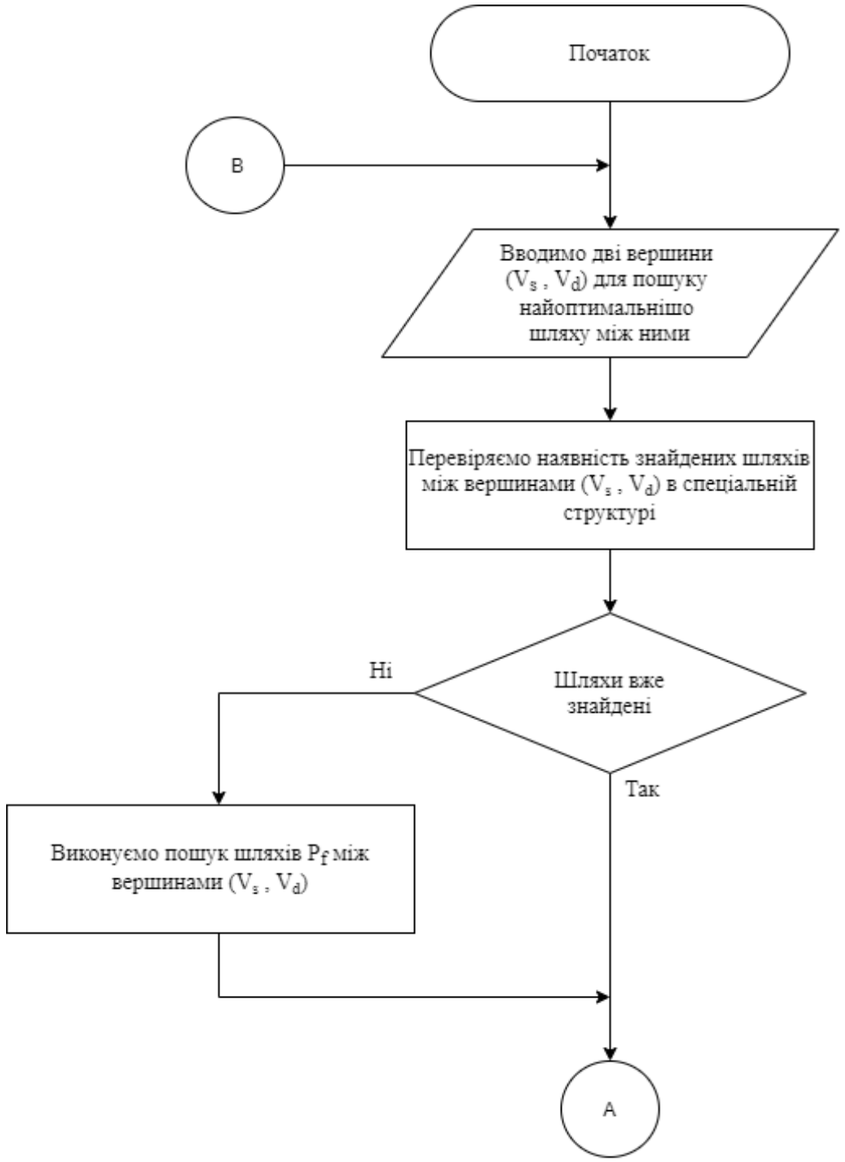
Спосіб конструювання трафіку з врахуванням рівномірного завантаження мережі

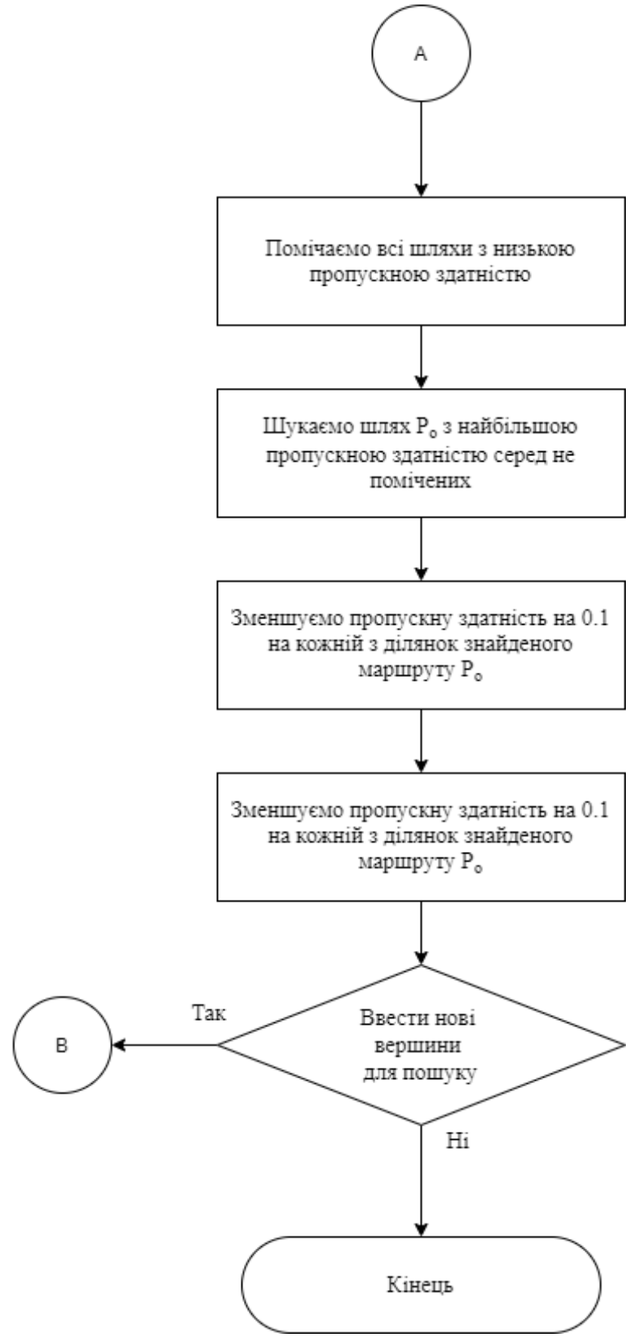
Принципова схема алгоритму

ІАЛЦ. 466454.004 Д1

Аркушів 1

Київ 2020 р





					ІАЛЦ.466454.004 Д1		
Зм.	*Арк.*	*№ докум.*	*Підпис*	*Дата*	*Спосіб конструювання трафіку з врахуванням рівномірного завантаження мережі Принципова схема алгоритму*	*Літ.*	*Маса* / *Масштаб*
Розроб.		*Савченко Ю.Ю.*					
Перевір.		*Калюжний О.О.*				*Арк.*	*Аркушів*
Н. контр.		*Сімоненко В.П.*			*Дипломна робота*	*КПІ ФІОТ кафедра ОТ гр. ІО-62*	
Затверд.							

ДОДАТОК 2

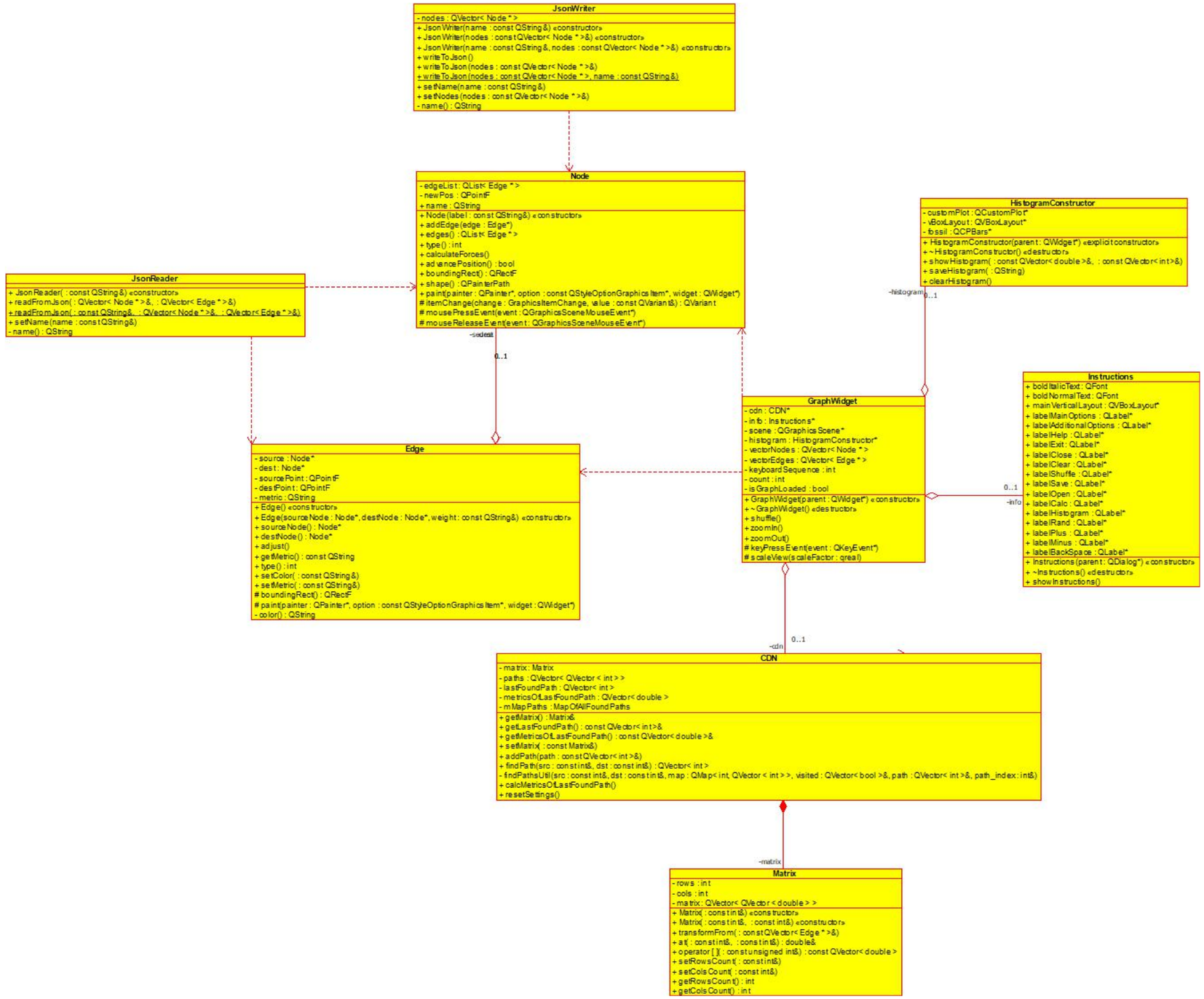
Спосіб конструювання трафіку з врахуванням рівномірного завантаження мережі

Функціональна схема (діаграма класів)

ІАЛЦ. 466454.005 Д2

Аркушів 1

Київ 2020 р



					ІАЛЦ.466454.005 Д2		
Зм.	Арк.	№ докум.	Підпис	Дата	Спосіб конструювання трафіку з врахуванням рівномірного завантаження мережі Функціональна схема (діаграма класів	Літ.	Маса / Масштаб
Розроб.		Савченко Ю.Ю.					
Перевір.		Калюжний О.О.				Арк.	Аркушів
Н. контр.		Сімоненко В.П.			Дипломна робота	КПІ ФІОТ кафедра ОТ гр. ІО-62	
Затверд.							

ДОДАТОК З

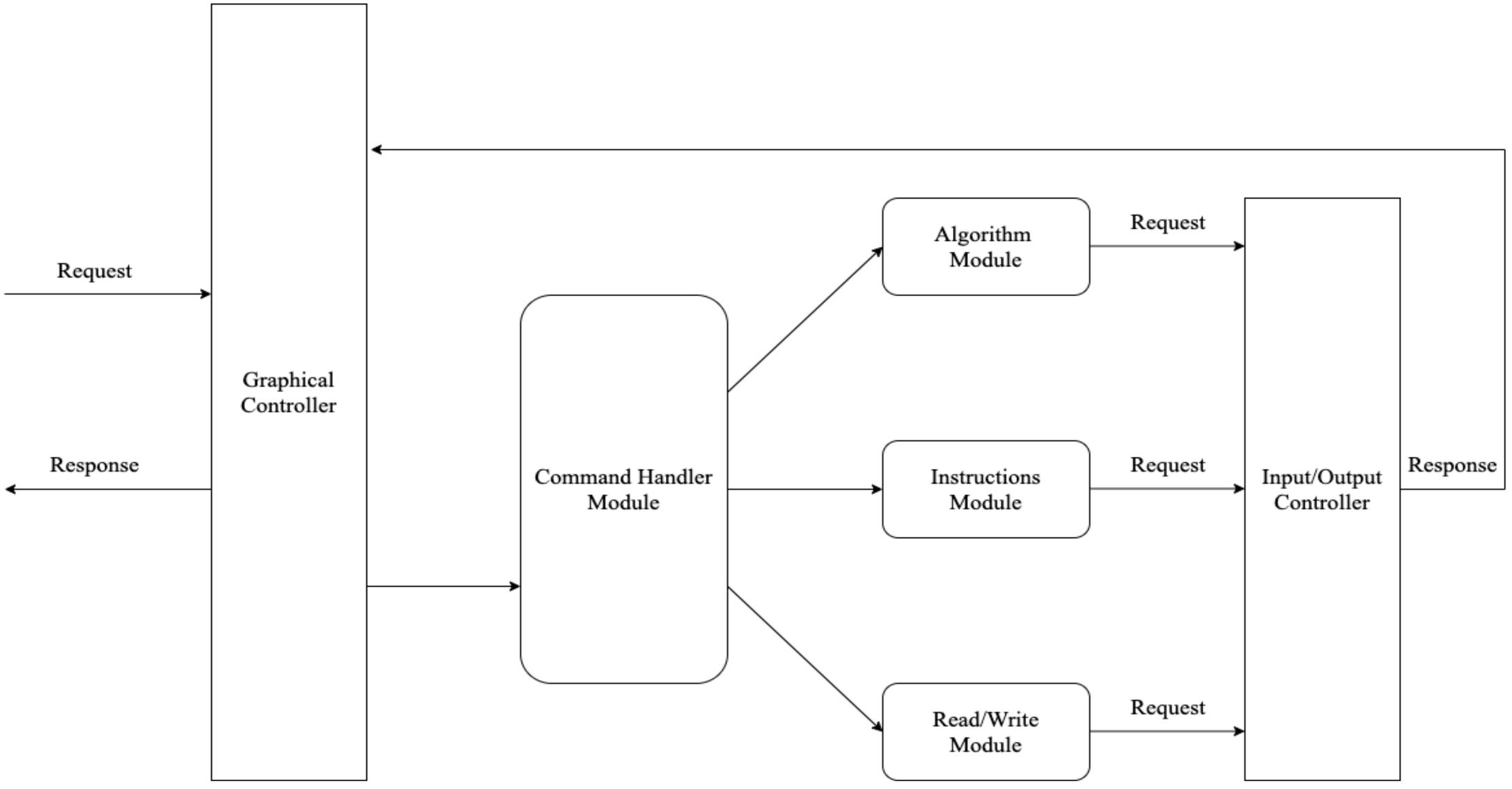
Спосіб конструювання трафіку з врахуванням рівномірного завантаження мережі

Структурна схема

ІАЛЦ. 466454.006 Д3

Аркушів 1

Київ 2020 р



					ІАЛЦ.466454.006 ДЗ	
Зм.	Арк.	№ докум.	Підпис	Дата	Спосіб конструювання трафіку з врахуванням рівномірного завантаження мережі Структурна схема	Літ. / Маса / Масштаб
Розроб.		Савченко Ю.Ю.				
Перевір.		Калюжний О.О.				Арк. / Аркушів
Н. контр.		Сімоненко В.П.			Дипломна робота	КПІ ФІОТ кафедра ОТ гр. ІО-62
Затверд.						

ДОДАТОК 4

Спосіб конструювання трафіку з врахуванням рівномірного завантаження мережі

Текст програми

ІАЛЦ. 466454.007.Д4

Аркушів 23

Київ – 2020 року

```
#ifndef CDN_H
#define CDN_H

#include <iostream>
#include <QMap>
#include <QPair>
#include <QVector>
#include <QQueue>
#include <QDebug>
#include <QElapsedTimer>
#include <cmath>

#include "../../headers/Graph/edge.h"
#include "../../headers/CDN/matrix.h"

using MapOfAllFoundPaths = QMap<QPair<int, int>, QVector<QPair<bool, QVector<int>>>>;

//!
//! \brief The CDN class \n
//! \b CDN - class which presents \b Content \b Delivery \b Network \n
//! \b CDN allows to save \b matrix of \b hosts and also gives access
//! calculate the shortest \e path of delivery \e packages \n
//!
class CDN
{
public:
    CDN() = default;                            //!< A \e constructor
    ~CDN() = default;                           //!< A \e destructor

private:
    Matrix matrix;                                  //!< A \e variable \b matrix
for saving \b hosts

    QVector<QVector<int>> paths;                    //!< A \e variable for saving
all \a paths of \b graph between \a source node and \a dest node
    QVector<int> lastFoundPath;                     //!< A \e variable for saving
the last found \b path
```

```
    QVector<double> metricsOfLastFoundPath;             //!< A \e variable for saving
\e list of \b metrics for the last found \b path


    MapOfAllFoundPaths mMapPaths;                       //!< A \e variable for saving
\e map of \b all found \b paths


public:

    Matrix& getMatrix();                                        //!< A \e function
for getting \b matrix


    const QVector<int> & getLastFoundPath();                    //!< A \e function
for getting the last found \b path

    const QVector<double> & getMetricsOfLastFoundPath();        //!< A \e function
for getting \b metrics of the last found \b path


public:

    void setMatrix(const Matrix &);                             //!< A \e method for
saving \b matrix


public:

    void addPath(const QVector<int> &path);                     //!< A \e method for
saving one \a path of \b graph

    QVector<int> findPath(const int &src, const int &dst);      //!< A \e method for
searching neighbors \a nodes in \b graph


private:

    //!

    //! \brief findPathsUtil

    //! \param src

    //! \param dst

    //! \param map

    //! \param visited

    //! \param path

    //! \param path_index

    //!

    void findPathsUtil(const int &src, const int &dst, QMap<int, QVector<int>> map,
QVector<bool> &visited, QVector<int> &path, int &path_index);


public:

    //!<
```

```
    //!< \brief calcMetricsOfLastFoundPath
    //!<
    void calcMetricsOfLastFoundPath();

public:
    //!<
    //!< \brief resetSettings
    //!<
    void resetSettings();
};

#endif // CDN_H

#ifndef MATRIX_H
#define MATRIX_H

#include <QVector>
#include <QDebug>
#include <QSize>

#include "../../headers/Graph/edge.h"
#include "../../headers/Graph/node.h"

//!
//! \brief The Matrix class\n
//! \b Matrix - presents mathematic type\n
//! Main target for \b matrix it's saving all \b hosts
//! which has beed created via \a Graph \a Builder \n
//!
class Matrix
{
public:
    Matrix(const int &);                        //!< A particular \e constructor with
parameter \b nodes
    Matrix(const int &, const int &);           //!< A particular \e constructor with
parameters \b rows and \b cols
    Matrix() = default;                         //!< A \e basic \a constructor
```

```
    ~Matrix() = default;                          //!< A \e destructor


private:

    int rows;                                      //!< A \e count of \b rows for
matrix

    int cols;                                      //!< A \e count of \b columns for
matrix

    QVector<QVector<double>> matrix;              //!< A \e variable which presents
this type \b matrix


public:

    void transformFrom(const QVector<Edge *> &);    //!< A \e method for transforming
into \b matrix


public:

    double &at(const int &, const int &);                          //!< A \e
function for getting or changing value from \b matrix

    const QVector<double> operator[](const unsigned int &) const;   //!< Overridden
\e operator for getting \b vector by position


public:

    void setRowsCount(const int &);               //!< A \e method which allows setup
value for \b row

    void setColsCount(const int &);               //!< A \e method which allows setup
value for \b col


public:

    int getRowsCount();                            //!< A \e method which allows give
value of \b row

    int getColsCount();                            //!< A \e method which allows give
value of \b col

};


#endif // MATRIX_H


#ifndef EDGE_H

#define EDGE_H


#include <QGraphicsItem>
```

```
class Node;

class Edge : public QGraphicsItem
{
public:
    Edge();
    Edge(Node *sourceNode, Node *destNode, const QString &weight);

    Node *sourceNode() const;
    Node *destNode() const;

    void adjust();
    const QString getMetric();

    enum { Type = UserType + 2 };
    int type() const override { return Type; }

    void setColor(const QString &);
    void setPenWidth(const int &);
    void setMetric(const QString &);

protected:
    QRectF boundingRect() const override;
    void paint(QPainter *painter, const QStyleOptionGraphicsItem *option, QWidget
*widget) override;

private:
    Node *source, *dest;

    QPointF sourcePoint;
    QPointF destPoint;
    QString metric;

    QString color {"gray"};
    int penWidth {1};
};
```

```
#endif // EDGE_H

#ifndef GRAPHWIDGET_H
#define GRAPHWIDGET_H

#include <QGraphicsView>
#include <QDebug>
#include <QFont>
#include <QLabel>
#include <QInputDialog>
#include <QFileDialog>
#include <QCoreApplication>
#include <QDateTime>
#include <QTime>

#include "../Parse/jsonwriter.h"
#include "../Parse/jsonreader.h"
#include "../CDN/cdn.h"
#include "../Info/instructions.h"
#include "../Info/histogramconstructor.h"

class Node;
class Edge;

class GraphWidget : public QGraphicsView
{
    Q_OBJECT

public:
    GraphWidget(QWidget *parent = 0);
    virtual ~GraphWidget();

public slots:
    void shuffle();
    void zoomIn();
```

```
    void zoomOut();

protected:
    void keyPressEvent(QKeyEvent *event) override;
#if QT_CONFIG(wheelevent)
    void wheelEvent(QWheelEvent *event) override;
#endif

    void scaleView(qreal scaleFactor);

private:
    CDN *cdn;
    Instructions *info;
    QGraphicsScene *scene;
    HistogramConstructor *histogram;

    QVector<Node *> vectorNodes;
    QVector<Edge *> vectorEdges;

    int keyboardSequence = 0;
    int count;
    bool isGraphLoaded;
};

#endif // GRAPHWIDGET_H

#ifndef NODE_H
#define NODE_H

#include <QGraphicsItem>
#include <QList>

class Edge;
class GraphWidget;
QT_BEGIN_NAMESPACE
class QGraphicsSceneMouseEvent;
```

```
QT_END_NAMESPACE

class Node : public QGraphicsItem
{
public:
    Node(const QString &label);

    void addEdge(Edge *edge);
    QList<Edge *> edges() const;

    enum { Type = UserType + 1 };
    int type() const override { return Type; }

    void calculateForces();
    bool advancePosition();

    QRectF boundingRect() const override;
    QPainterPath shape() const override;
    void paint(QPainter *painter, const QStyleOptionGraphicsItem *option, QWidget
*widget) override;

protected:
    QVariant itemChange(GraphicsItemChange change, const QVariant &value) override;

    void mousePressEvent(QGraphicsSceneMouseEvent *event) override;
    void mouseReleaseEvent(QGraphicsSceneMouseEvent *event) override;

private:
    QList<Edge *> edgeList;
    QPointF newPos;

public:
    QString name;
};

#endif // NODE_H
```

```
#ifndef HISTOGRAM_CONSTRUCTOR_H
#define HISTOGRAM_CONSTRUCTOR_H

#include <QWidget>
#include <QVBoxLayout>
#include <QMessageBox>
#include <QLabel>
#include <QVector>
#include <QPen>

/* including of the qcustomplot class */
#include "../QCustomPlot/qcustomplot.h"

//!
//! \brief The HistogramConstructor class \n
//! \b HistogramConstructor - class which presents \a mechanism for creating \b
histograms \n
//! \b HistogramConstructor allows to create a histogram and work with it by simple
functions \n
//!
class HistogramConstructor : public QWidget
{

    Q_OBJECT

public:
    explicit HistogramConstructor(QWidget *parent = nullptr);       //!< The explicit
\e constructor
    virtual ~HistogramConstructor();                                //!< The \e
virtual \e destructor

private:
    QCustomPlot *customPlot;        //!< The \e object which presents scene of \a
QCustomPlot
    QVBoxLayout *vBoxLayout;        //!< The \e object which presents \b Vertical \b
Layout
    QCPBars *fossil;                //!< The \e plottable representing a \b bar \b
chart in a \b plot.
```

```
public:

    void showHistogram(const QVector<double> &, const QVector<int> &);      //!< The
\e method which allows to show the final \b histogram

    void saveHistogram(QString);                                            //!< The
\e method which allows to save the \b histogram as \a PNG

    void clearHistogram();                                                  //!< The
\e method which allows to clear all settings for \b histogram

};


#endif // HISTOGRAM_CONSTRUCTOR_H

#ifndef INSTRUCTIONS_H

#define INSTRUCTIONS_H


#include <QDialog>

#include <QString>

#include <QLabel>

#include <QFont>

#include <QVBoxLayout>


const QString MAIN_OPTIONS="Controlling of the application is available by next
commands:\n\n";


const QString HELP="command allows to print instructions of using the application.";

const QString EXIT="command allows to exit from the application.";

const QString CLOSE="command allows to close the graph.";

const QString CLEAR="command allows to reset all found paths on the graph.";

const QString SHUFFLE="command allows to shuffle all nodes on the graph.";

const QString SAVE="command allows to save all changes on the scene.";

const QString OPEN="command allows to upload the saved scene.";

const QString CALC="command allows to find the best path on the graph (enter nodes
through comma: 1,7).";

const QString HISTOGRAM="command allows to create histogram which demonstrates
metrics.";

const QString RAND="command allows to randomize whole metric on the graph.";


const QString ADDITIONAL_OPTIONS="\n\nControlling of the application by additional
commands:\n\n";
```

```
const QString PLUS="allows to zoom in the graph.";
const QString MINUS="allows to zoom out the graph.";
const QString BACKSPACE="allows to delete uncorrect word that you wrote.";


//!
//! \brief The Instructions class\n
//! \b Instructions it's special class which allows
//! print \a rules how to control an \a application.
//!
class Instructions : public QDialog {
public:
    Instructions(QDialog *parent = nullptr);    //!< A typicall \e constructor
    ~Instructions();                            //!< A typicall \e destructor


public:
    void showInstructions();                    //!< A \e functions which allows to
start an \e execution of \a window with \b Instructions


public:
    QFont boldItalicText;                       //!< A combination of \e Bold and \e
Italic \b Font for \a text
    QFont boldNormalText;                       //!< A combination of \e Bold and \e
Normal \b Font for \a text


    QVBoxLayout *mainVerticalLayout;            //!< A \b mainVerticalLayout \a
variable which allows to save other \e layouts


    QLabel *labelMainOptions;                   //!< A \a label for demonstration \b
MainOptions
    QLabel *labelAdditionalOptions;             //!< A \a label for demonstration \b
AdditionalOptions


    QLabel *labelHelp;                          //!< A \a label for demonstration \b
Help \e command
    QLabel *labelExit;                          //!< A \a label for demonstration \b
Exit \e command
    QLabel *labelClose;                         //!< A \a label for demonstration \b
Close \e command
    QLabel *labelClear;                         //!< A \a label for demonstration \b
Clear \e command
```

```
    QLabel *labelShuffle;                       //!< A \a label for demonstration \b
Shuffle \e command

    QLabel *labelSave;                          //!< A \a label for demonstration \b
Save \e command

    QLabel *labelOpen;                          //!< A \a label for demonstration \b
Open \e command

    QLabel *labelCalc;                          //!< A \a label for demonstration \b
Calc \e command

    QLabel *labelHistogram;                     //!< A \a label for demonstration \b
Histogram command

    QLabel *labelRand;                          //!< A \a label for demonstration \b
Rand command


    QLabel *labelPlus;                          //!< A \a label for demonstration \b
+ \e command

    QLabel *labelMinus;                         //!< A \a label for demonstration \b
- \e command

    QLabel *labelBackSpace;                     //!< A \a label for demonstration \b
<- \e command

};


#endif // INSTRUCTIONS_H


#ifndef JSONREADER_H

#define JSONREADER_H


#include <QFile>

#include <QString>

#include <QDebug>

#include <QJsonDocument>

#include <QJsonArray>

#include <QJsonObject>

#include <QJsonValue>

#include <QMessageBox>


#include "../Graph/node.h"

#include "../Graph/edge.h"


//!

//! \brief The JsonReader class\n
```

```
//! \b JsonReader it's special class which allows

//! read configuration from \a json \a file for

//! \b graph and print it to \a scene.

//!

class JsonReader

{

public:

    JsonReader(const QString &);            //!< A \e constructor for saving \b name
of file

    JsonReader() = default;                 //!< A default \e constructor

    ~JsonReader() = default;                //!< A default \e destructor


public:

    void readFromJson(QVector<Node *> &, QVector<Edge *> &);
//!< A \e method for reading from \a json \a file

    static void readFromJson(const QString &, QVector<Node *> &, QVector<Edge *> &);
//!< A \e method for reading from \a json \a file with particular \b name


public:

    void setName(const QString &name);      //!< A method for setting \b name of file
into variable


private:

    QString name { "config.json" };         //!< A \b name of \a json \a file

};


#endif // JSONREADER_H

#ifndef JSONWRITER_H

#define JSONWRITER_H


#include <QFile>

#include <QString>

#include <QDebug>

#include <QJsonDocument>

#include <QJsonObject>

#include <QJsonValue>

#include <QMessageBox>
```

```
#include "../Graph/node.h"

#include "../Graph/edge.h"


//!

//! \brief The JsonWriter class\n

//! \b JsonWriter it's special \e class which allows

//! write configuration of graph to \e json file.

//!

class JsonWriter

{

public:

    JsonWriter(const QString &name);                                    //!< A \e
constructor which save \b name of \a file

    JsonWriter(const QVector<Node *> &nodes);                           //!< A \e
constructor which save list of \b nodes

    JsonWriter(const QString &name, const QVector<Node *> &nodes);  //!< A \e
constructor which save \b name of \a file and list of \b nodes

    JsonWriter() = default;                                             //!< A \e default
\e constructor

    ~JsonWriter() = default;                                            //!< A \e default
\e destructor


public:

    void writeToJson();                                                         //!<
A \e method for writing to \a json \a file

    void writeToJson(const QVector<Node *> &nodes);                             //!<
A \e method for writing to \a json \a file by particular list of \b nodes

    static void writeToJson(const QVector<Node *> nodes, const QString &name);  //!<
A \e method for writing to \a json \a file by particular list of \b nodes and certain
\b name of \a file


public:

    void setName(const QString &name);              //!< A \e method for setting \b
name of \a file into variable

    void setNodes(const QVector<Node *> &nodes);    //!< A \e method for setting \b
nodes of \b graph into variable


private:

    QString name { "config.json" };     //!< A \b name of \a file for writing

    QVector<Node *> nodes;              //!< A variable for saving \b nodes of \b
graph
```

```
};


#endif // JSONWRITER_H


#include "../../headers/CDN/cdn.h"


/* implementation of function which allows to get the matrix */
Matrix &CDN::getMatrix()
{
    return this->matrix;
}


/* implementation of the function which allows to get the last found path */
const QVector<int> & CDN::getLastFoundPath()
{
    return this->lastFoundPath;
}


/* implementation of the function which allows to get the metrics of the last found
path */
const QVector<double> & CDN::getMetricsOfLastFoundPath()
{
    return this->metricsOfLastFoundPath;
}


/* implementation of method for saving matrix */
void CDN::setMatrix(const Matrix &matrix)
{
    this->matrix = matrix;
}


/* implementation of method for adding of path to all paths */
void CDN::addPath(const QVector<int> &path)
{
    paths.push_back(path);
}
```

```
/* implementation of method which allows to find all neighbors for all nodes */
QVector<int> CDN::findPath(const int &src, const int &dst)
{
    if (!mMapPaths.contains(QPair<int, int> {src, dst}) &&
!mMapPaths.contains(QPair<int, int> {dst, src})) {
        paths.clear();

        // Mark all the vertices as not visited
        QVector<bool> visited(matrix.getRowsCount(), false);

        // Create an array to store paths
        QVector<int> path(matrix.getRowsCount());
        int path_index = 0;

        // Create an map which stores a lists of neighbors nodes
        QMap<int, QVector<int>> neighborsNodes;

        for (int i=0; i < matrix.getRowsCount(); i++)
            neighborsNodes[i] = QVector<int> ();

        for (int i=0; i < matrix.getRowsCount(); i++) {
            for (int j=0; j < matrix.getColsCount(); j++) {
                if (matrix.at(i, j) > 0) {
                    neighborsNodes[i].append(j);
                    neighborsNodes[j].append(i);
                }
            }
        }

        // Call the recursive helper function to find all paths
        findPathsUtil(src, dst, neighborsNodes, visited, path, path_index);

        QVector<QPair<bool, QVector<int>>> markedPaths;

        for (int i=0; i < paths.size(); i++)
```

```
            markedPaths.append(QPair<bool, QVector<int>> {false, paths[i]});

        if (src < dst)
            mMapPaths[QPair<int, int> {src, dst}] = markedPaths;

        else if (dst < src)
            mMapPaths[QPair<int, int> {dst, src}] = markedPaths;
    }

    // Map for saving all paths and their metric
    QMap<int, double> matchPathMetric;
    QVector<QPair<bool, QVector<int>>> foundPaths;

    if (src < dst)
        foundPaths = mMapPaths[QPair<int, int> {src, dst}];

    else if (dst < src)
        foundPaths = mMapPaths[QPair<int, int> {dst, src}];

    for (int i=0; i < foundPaths.size(); i++) {
        for (int j=0; j < foundPaths[i].second.size()-1; j++) {
            if (matrix.at(foundPaths[i].second[j], foundPaths[i].second[j+1]) > 0.0
&&
                matrix.at(foundPaths[i].second[j], foundPaths[i].second[j+1]) <= 0.1)
{
                foundPaths[i].first = true;
                break;
            }

            else if (matrix.at(foundPaths[i].second[j+1], foundPaths[i].second[j]) >
0.0 &&
                matrix.at(foundPaths[i].second[j+1], foundPaths[i].second[j]) <= 0.1)
{
                foundPaths[i].first = true;
                break;
            }

        }
```

```
    }

    // Looking for the best optimality path for delivery
    for (int i=0; i < foundPaths.size(); i++) {
        if (foundPaths[i].first)
            continue;

        double sum = 0;

        // Count metric like difference between maximal value and existing
        for (int j=0; j < foundPaths[i].second.size()-1; j++) {
            // If metric value places upper than diagonal
            if (matrix.at(foundPaths[i].second[j], foundPaths[i].second[j+1])) {
                sum += 1 - matrix.at(foundPaths[i].second[j],
foundPaths[i].second[j+1]);
            }
            // If metric value places under diagonal
            else if (matrix.at(foundPaths[i].second[j+1], foundPaths[i].second[j])) {
                sum += 1 - matrix.at(foundPaths[i].second[j+1],
foundPaths[i].second[j]);
            }
        }

        matchPathMetric[i] = sum;
    }

    double min = INT_MAX;
    int lastFoundPathIndex = -1;

    for (const auto &key : matchPathMetric.keys())
        if (matchPathMetric[key] < min)
            min = matchPathMetric[key];

    for (int i=0; i < foundPaths.size(); i++) {
        if (matchPathMetric[i] == min && !foundPaths[i].first) {
            // saving path as last found path
            lastFoundPathIndex = i;
```

```
            this->lastFoundPath = foundPaths[i].second;
            break;
        }
    }

    if (lastFoundPathIndex == -1)
        return QVector<int> {};

    for (int i=0; i < foundPaths[lastFoundPathIndex].second.size()-1; i++) {
        if
(matrix[foundPaths[lastFoundPathIndex].second[i]][foundPaths[lastFoundPathIndex].seco
nd[i+1]] > 0.1) {
            matrix.at(foundPaths[lastFoundPathIndex].second[i],
foundPaths[lastFoundPathIndex].second[i+1]) =

matrix[foundPaths[lastFoundPathIndex].second[i]][foundPaths[lastFoundPathIndex].secon
d[i+1]]-0.1;
        }

        else if
(matrix[foundPaths[lastFoundPathIndex].second[i+1]][foundPaths[lastFoundPathIndex].se
cond[i]] > 0.1) {
            matrix.at(foundPaths[lastFoundPathIndex].second[i+1],
foundPaths[lastFoundPathIndex].second[i]) =

matrix[foundPaths[lastFoundPathIndex].second[i+1]][foundPaths[lastFoundPathIndex].sec
ond[i]]-0.1;
        }
    }

    return lastFoundPath;
}

/* implementation of method which is looking for all paths in graph between source
node and destonation node */
void CDN::findPathsUtil(const int &src, const int &dst, QMap<int, QVector<int>> map,
QVector<bool> &visited, QVector<int> &path, int &path_index)
{
    // Mark the current node and store it in path[]
    visited[src] = true;
    path[path_index] = src;
```

```
    path_index++;

    // If current vertex is same as destination, then add path to paths
    if (src == dst) {
        QVector<int> newPath;

        for (int i=0; i < path_index; i++)
            newPath.append(path[i]);

        addPath(newPath);
    }

    // If current vertex is not destination
    else {
        // Recur for all the vertices adjacent to current vertex
        QVector<int>::iterator itr;

        for (itr=map[src].begin(); itr != map[src].end(); ++itr)
            if (!visited[*itr])
                findPathsUtil(*itr, dst, map, visited, path, path_index);
    }

    // Remove current vertex from path[] and mark it as unvisited
    path_index--;
    visited[src] = false;
}

/* implementation of the method that allows to calculate metrics for last found path
*/
void CDN::calcMetricsOfLastFoundPath()
{
    this->metricsOfLastFoundPath.clear();

    for (int i=0; i < lastFoundPath.size()-1; i++) {
        if (matrix[lastFoundPath[i]][lastFoundPath[i+1]] > 0)

metricsOfLastFoundPath.append(matrix[lastFoundPath[i]][lastFoundPath[i+1]]);
```

```
        if (matrix[lastFoundPath[i+1]][lastFoundPath[i]] > 0)

metricsOfLastFoundPath.append(matrix[lastFoundPath[i+1]][lastFoundPath[i]]);
    }
}

/* implementation of the method that allows to reset all settings for cdn controller
*/
void CDN::resetSettings()
{
    this->paths.clear();
    this->lastFoundPath.clear();
    this->metricsOfLastFoundPath.clear();
    this->mMapPaths.clear();
}

#include "../../headers/CDN/matrix.h"

/* implementation of explicit constructor, which also gets count of nodes */
Matrix::Matrix(const int &nodes)
    : rows(nodes), cols(nodes)
{
    // creating matrix
    for (int i=0; i < rows; i++) {
        QVector<double> tmp(cols, 0);
        matrix.append(tmp);
    }
}

/* implementation of explicit constructor, which also gets count of rows and cols */
Matrix::Matrix(const int &rows, const int &cols)
    : rows(rows), cols(cols)
{
    // creating matrix
    for (int i=0; i < rows; i++) {
        QVector<double> tmp(cols, 0);
```

```
        matrix.append(tmp);
    }
}

/* implementation of function which allows transform typical list with edges into
numeric matrix */
void Matrix::transformFrom(const QVector<Edge *> &edges)
{
    // creating matrix
    for (int i=0; i < rows; i++) {
        QVector<double> tmp(cols, 0);
        matrix.append(tmp);
    }

    // filling matrix by values of metric from vector edges
    for (int i=0; i < edges.size(); i++)
        matrix[edges[i]->sourceNode()->name.toInt()-1][edges[i]->destNode()-
>name.toInt()-1] = edges[i]->getMetric().toDouble();
}

/* implementation of function which allows get access till particular value from
matrix */
double &Matrix::at(const int &i, const int &j)
{
    return this->matrix[i][j];
}

/* implementation of overridden operator which allows return vector from particular
position */
const QVector<double> Matrix::operator[](const unsigned int &i) const
{
    return this->matrix[i];
}

/* implementation of method for setting value into variable row */
void Matrix::setRowsCount(const int &rows)
{
    this->rows = rows;
```

```
}

/* implementation of method for setting value into variable col */
void Matrix::setColsCount(const int &cols)
{
    this->cols = cols;
}

/* implementation of method for getting value from variable row */
int Matrix::getRowsCount()
{
    return this->rows;
}

/* implementation of method for getting value from variable col */
int Matrix::getColsCount()
{
    return this->cols;
}
```